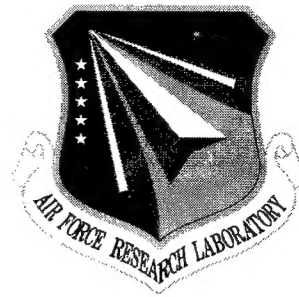AFRL-IF-RS-TR-2001-47
Final Technical Report
April 2001

HOL2GDT A FORMAL VERIFICATION-BASED DESIGN METHODOLOGY

Syracuse University

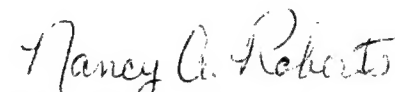**Anand Chavan, Byoung Woo Min, and Shiu-Kai Chin**

20010607 003

AIR FORCE RESEARCH LABORATORY
INFORMATION DIRECTORATE
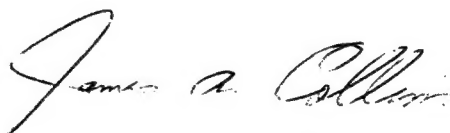**ROME RESEARCH SITE**
ROME, NEW YORK

This report has been reviewed by the Air Force Research Laboratory, Information Directorate, Public Affairs Office (IFOIPA) and is releasable to the National Technical Information Service (NTIS). At NTIS it will be releasable to the general public, including foreign nations.

AFRL-IF-RS-TR-2001-47 has been reviewed and is approved for publication.

APPROVED: *Nancy A. Roberts*

NANCY A. ROBERTS
Project Engineer

FOR THE DIRECTOR: *James A. Collins*

JAMES A. COLLINS, Acting Chief
Information Technology Division
Information Directorate

REPORT DOCUMENTATION PAGE		*Form Approved* *OMB No. 0704-0188*

Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington Headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188), Washington, DC 20503.

1. AGENCY USE ONLY *(Leave blank)*	2. REPORT DATE APRIL 2001	3. REPORT TYPE AND DATES COVERED Final Sep 97 - Mar 99	
4. TITLE AND SUBTITLE HOL2GDT A FORMAL VERIFICATION-BASED DESIGN METHODOLOGY			5. FUNDING NUMBERS C - F30602-97-C-0310 PE - 62702F PR - 5581 TA - 27 WU - PT
6. AUTHOR(S) Anand Chavan, Byoung Woo Min, and Shiu-Kai Chin			

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Syracuse University Case Center Syracuse NY 13244	8. PERFORMING ORGANIZATION REPORT NUMBER N/A
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES) Air Force Research Laboratory/IFTD 525 Brooks Road Rome NY 13441-4505	10. SPONSORING/MONITORING AGENCY REPORT NUMBER AFRL-IF-RS-TR-2001-47

11. SUPPLEMENTARY NOTES
Air Force Research Laboratory Project Engineer: Nancy Roberts/IFTD/(315) 330-3566

12a. DISTRIBUTION AVAILABILITY STATEMENT APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.	12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*
HOL2GDT is a VLSI design methodology. It starts with a design implementation description that is formally verified using the Higher Order Logic (HOL) theorem prover. This implementation description is translated into a hardware description language model by using a HOL2GDT compiler, and with this model a physical design layout is generated by using IC design placement and routing tools in Mentor Graphic's Generator Design Technology (GDT) package. Thus, the final IC layout is generated from a formally verified description. This document illustrates the design methodology in detail to serve as a manual for the HOL2GDT system. It covers (1) how to define formal implementation descriptions of the hardware design, (2) how to translate implementation descriptions into L language schematic generator models, and (3) how to get physical IC layouts from schematic models. A complete example of an n-bit Serial Multiplier design is used to illustrate the HOL2GDT design methodology.

14. SUBJECT TERMS Verified Design, Higher Order Logic (HOL), VLSI design, Hardware Modeling			15. NUMBER OF PAGES 126
			16. PRICE CODE
17. SECURITY CLASSIFICATION OF REPORT UNCLASSIFIED	18. SECURITY CLASSIFICATION OF THIS PAGE UNCLASSIFIED	19. SECURITY CLASSIFICATION OF ABSTRACT UNCLASSIFIED	20. LIMITATION OF ABSTRACT UL

Standard Form 298 (Rev. 2-89) (EG)
Prescribed by ANSI Std. 239.18
Designed using Perform Pro, WHS/DIOR, Oct 94

Abstract

HOL2GDT is a VLSI design methodology. It starts with a *design implementation description* that is formally verified using the Higher Order Logic (HOL) theorem prover. This implementation description is translated into a hardware description language model by using a HOL2GDT compiler, and with this model a physical design layout is generated by using IC design placement and routing tools in Mentor Graphic's Generator Design Technology (GDT) package. Thus the final IC layout is generated from a formally verified description. This document illustrates the design methodology in detail to serve as a manual for the HOL2GDT system. It covers: (1) how to define formal implementation descriptions of the hardware design, (2) how to translate implementation descriptions into L language schematic generator models, and (3) how to get physical IC layouts from schematic models. A complete example of an n-bit Serial Multiplier design is used to illustrate the HOL2GDT design methodology.

Contents

iv

List of Figures

List of Tables

1   Introduction to HOL2GDT

HOL2GDT methodology uses the language of higher-order logic supported by the HOL, which is used to obtain an implementation description of a system as well as to verify the design. The HOL implementation description is then compiled into an L language [7] schematic generator model by the HOL2GDT compiler. The Mentor Graphics GDT tool suite is used for the placement and routing of the generator models to create a physical design layout. Next, the GDT layout is extracted into a Caltech Intemediate Format (CIF) netlist [6]. The CIF netlist is ported to a VLSI layout editor, MAGIC, to perform full chip integration and assembly. The CIF netlist translation from GDT to MAGIC is performed by the L2MCIF compiler.

HOL2GDT methodology provides a front-end/back-end design framework by developing strategic compilers to integrate the HOL formal verification environment with the Mentor Graphics placement and routing tools and the MAGIC chip assembly layout editor. The HOL2GDT methodology has been successfully used to fabricate an 8-bit serial pipelined multiplier chip.

HOL is a theorem prover that is used for formal verification. It provides a language syntax and semantics with which to embed the design specification and implementation description into the HOL environment. Using HOL, a correctness theorem which asserts that the implementation description of the system implies or is equivalent to the specification description of the system is established and proved.

Mentor Graphics GDT is a sophisticated IC design tool suite that provides schematic module generation, placement-and-routing tools, and a multi-level mixed mode functional simulator. GDT's module generator supports *parameterized* and *hierarchical* design descriptions, and HOL also supports these two characteristics. For example, the hierarchical module generator corresponds to the hierarchical implementation description in HOL, allowing the verification of a system to be done in hierarchical order. GDT's parameterized module generator has the ability to simplify the design of regular hardware structures; however, regular structures can be defined in HOL and it is usually easy to describe hardware circuits using HOL. The merit of the hardware implementation described by HOL is that it can be used in varifying correctness of the design. Figure 1 illustrates the organization of HOL2GDT methodology.

The outline of the rest of this paper is as follows. In Section 2 the HOL implemetation description mechanism is reviewed. In Section 3 the GDT system is described. Since a HOL implementation description is to be translated into an L language model, Section 4 is dedicated to describing the L language. Section 5 explains how to build an actual L language code, and Section 6 explains the translation mechanism from HOL

1
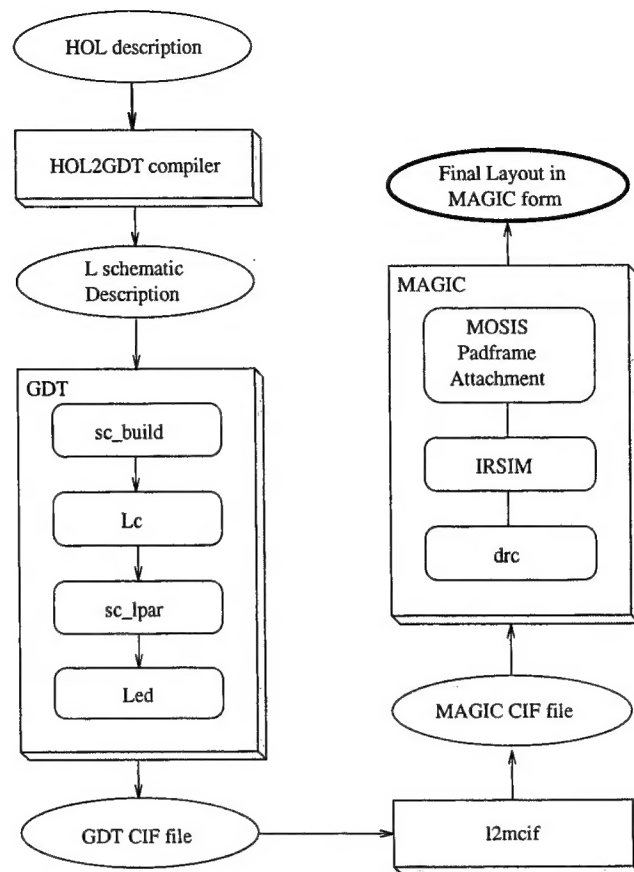
Figure 1: HOL2GDT Linking System

to the L language. Section 7 provides a complete example of an n-bit serial pipelined multiplier. Section 8 is concerned with the L2MCIF compiler, which converts a GDT CIF format into MAGIC CIF format for pad frame embedding. In Section 9, three simulators involved in this methodology are explained. The last section shows the final chip appearance with testing results. The appendix contains the actual *sml* code for the basic gates definition, the macro file needed in the GDT system, and the n-bit multiplier definition described in Section 7.

2

2 Hardware Description in HOL

This section explains how hardware components are represented by HOL notation. According to Tom Melham [1], most digital hardware components can be described using predicates, universal and existential quantifiers, conjunction, and recursion.

2.1 Relational Description

A digital hardware block is viewed as the input/output ports of the block that are seen from the outside and the combination of the basic logic gates such as AND, OR, XOR, and NOT. Mathematically, a predicate describes objects and the relationships between them. Thus, an n-ary predicate can specify a hardware block having n external ports by matching the predicate name with the block name and using the external port names as its parameters. With the help of quantifiers ("!" represents the universal quantifier, and "?" the existential quantifier in HOL), the properties of the input parameters of the predicates can be described. That is, universally quantified variables (parameters) are used to specify the external (input/output) signals to the block, and the existentially quantified variables are used to specify the internal signals (hidden lines used to interconnect subcomponents within the block).

The connections among the components in a hardware block are implemented by using *conjunctions* ("$\wedge$" in HOL notation). If two subcomponents are combined by conjunction and there are common parameters that are existentially quantified, then the components are connected through the line with the common variable name. Example 1 will clarify this concept.

Example 1. Relational Description of a Full Adder

Figure 2 shows the block diagram of a full adder, which consists of two XOR gates, two AND gates, and one OR gate. The symbols *a, b, and cin* are the input ports, and *sum* and *cout* are the output ports. There are three internal lines, *w1, w2,* and *w3*, for the interconnection between the subcomponents.

With FADDER as the predicate name for the full adder, and using the ordinary gate names predefined in HOL, a full adder can be described as follows:

3
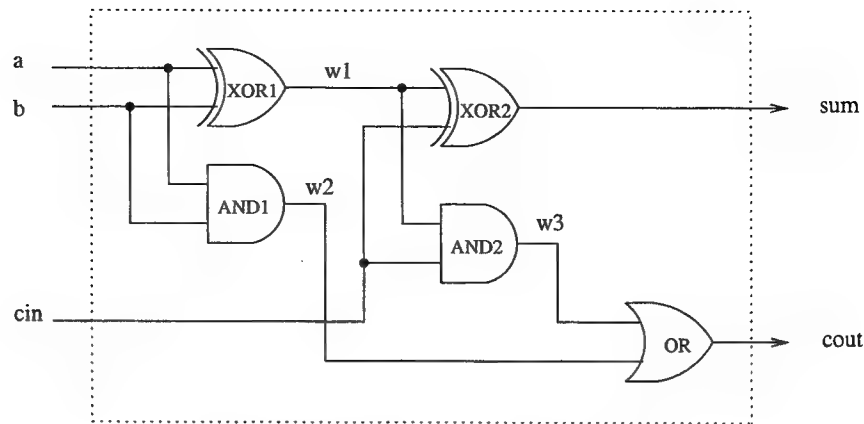
Figure 2: Diagram of Full Adder

```
! a b cin sum cout. FADDER (a, b, cin, sum, cout) =
  ? w1 w2 w3. xor(a, b, w1) /\
              and(a, b, w2) /\
              xor(w1, cin, sum) /\
              and(w1, cin, w3)  /\
              or (w2, w3, cout)
```

Here, "!" is the universal quantifier, and "?" represents the existential quantifier. The parameters *a, b, cin, sum,* and *cout* are matched with external port names, and *w1, w2,* and *w3* with the interconnecting lines hidden to the outside. The universal quantifier is used to specify the external signals, and the existential quantifier is used to specify the interconnecting lines in the HOL definition. The location of the parameters in the HOL definition conveys significant information. For example, in the XOR gate definition the first parameter represents the first input signal to the XOR gate, the second parameter represents the second input signal, and the last parameter represents the output signal. Thus, from the above full adder definition we can infer the fact that the output of the first XOR gate is connected to the first input of the second XOR gate. Finally, the subcomponents are conjoined each other by using the *conjunction* symbol, ∧.

4

2.2 Recursion

Recursion refers to a mathematical procedure of calling itself inside its definition. This recursion can be used in describing a circuit that may be constructed by interconnecting indentical components. A typical definition of a recursion consists of a *base case* and a *recursive* (or *inductive*) case definition. The base case specifies the primitive building block of a circuit and the recursive case defines how the primitive building block is added to increase the size of the circuit by one.

An n-bit ripple adder can be defined using recursion, and Example 2 illustrates this idea.

Example 2. An n-bit Ripple Adder

An n-bit ripple adder consists of n of FADDER as defined in Example 1. The block diagram of n-bit ripple adder is shown in Figure 3.
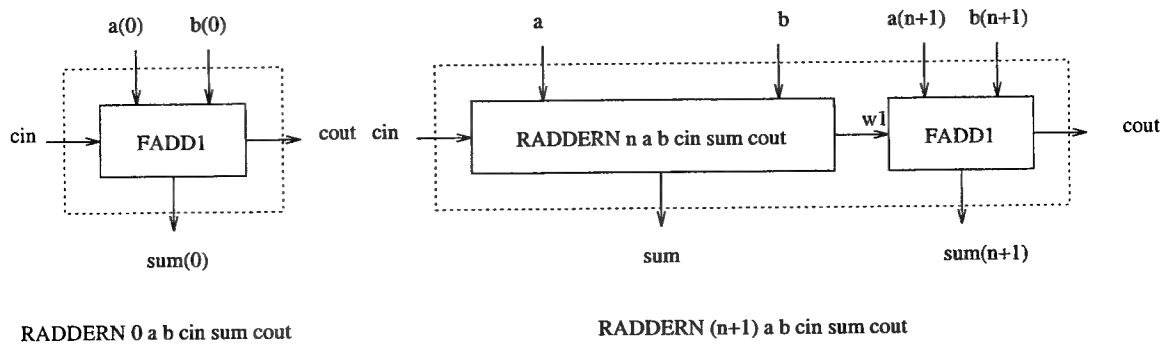
Figure 3: Diagram of an n-bit Rippler Adder

Below is a HOL description of the n-bit ripple adder where the predicate name is RADDERN.

5

RADDERN

```
|- ! a b cin sum cout. RADDERN 0 a b cin sum cout =
      FADDER ( a 0, b 0, cin, sum 0 cout)  /\
    ! n a b cin sum cout. RADDERN (n+1) a b cin sum cout =
    (? w1. RADDERN n a b cin sum w1  /\
            FADDER (a(n+1), b(n+1), w1, sum(n+1), cout))
```

The base case is simply the definition of a one-bit full adder. That is, if n is equal to zero, then the ripple adder becomes a one-bit full adder. The recursive case contains *RADDERN* itself and the additional one-bit full adder, *FADDER*. There is only one interconnecting line, *w1*, to connect *cout* terminal of the previous stage to *cin* terminal of the next stage.

Here, the recursive definition of an n bit ripple adder is a *simple recursion*, that is, the recursive case uses a primary building block, *FADDER*, which is not defined recursively. Sometimes circuit designs may involve definitions in which the base definition consists of several definitions and is defined recursively. This is called a **nested recursion** and is illustrated in Example 3.

Example 3. A Nested Recursive Description for an n-Bit S/P Multiplier

The block diagram for the n-bit serial/parallel multiplier is shown in Figure 4. The HOL definition of the multiplier is shown below:

SADDERN
```
|- (! clk a b co  sum reset. SADDERN 0 clk a b co sum reset =
      SADDER1(clk, a 0, b 0, co, sum, reset)) /\
    (! n clk a b co sum reset. SADDERN(n+1) clk a b co sum reset =
    (? w. SADDERN n clk a b co w reset /\
          SADDER1(clk, a(n+1), b(n+1), w, sum, reset)))
```

WPROD
```
|- (! mc m mout prod load reset clk.
      WPROD 0 mc m mout prod load reset clk =
```

6

SMULT 0 mc m mout s sout prod co reset load clk

SMULT n+1 mc m mout s sout prod co reset load clk
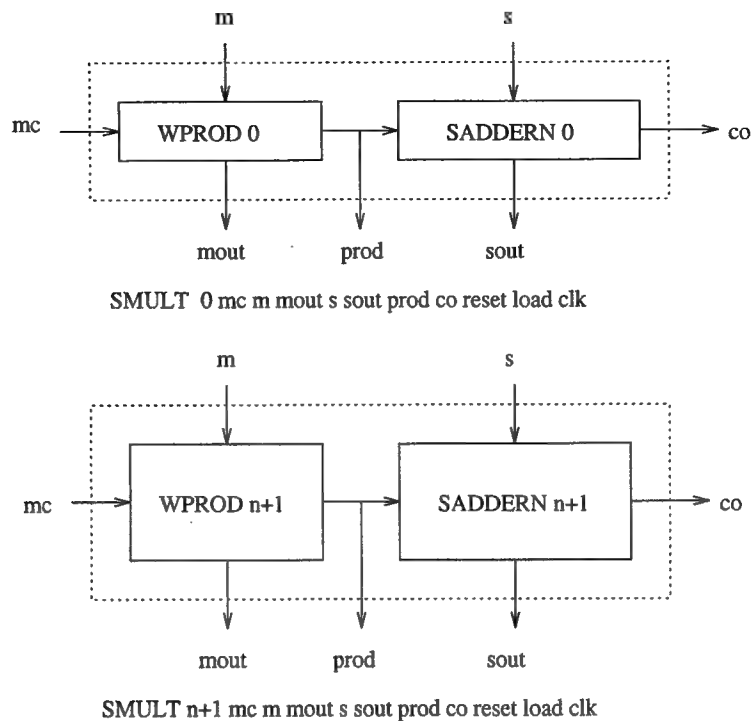
Figure 4: Diagram of n-Bit Serial/Parallel Multiplier

```
    WPROD1(mc 0, m, mout, prod 0, load, reset, clk)) /\
   (! n mc m mout prod load reset clk.
    WPROD (n+1) mc m mout prod load reset clk =
   (? w. WPROD n mc m w prod load reset clk /\
        WPROD1(m(n+1), w, mout, prod(n+1), load, reset, clk)))


SMULTN
|- (! mc m mout s prod co reset load clk.
     SMULTN 0 mc m mout s sout prod co reset load clk =
        WPROD 0 mc m mout prod load reset clk /\
        SADDERN 0 clk prod s co sout reset) /\
   (! n mc m mout s sout prod co reset load clk.
     SMULT (n+1) mc m mout s sout prod co reset load clk =
        WPROD (n+1) mc m mout prod load reset clk /\
        SADDERN (n+1) clk prod s co sout reset)
```

7

The definition of *SMULTN* not only contains two building blocks, *WPROD* and *SADDERN*, but also is defined recursively. That is, *WPROD* is defined using basic building cells *WPROD1* and *SADDERN*. Thus *SMULTN* is defined as being nested recursion. The current version of the HOL2GDT compilier cannot automatically generate layouts for descriptions with nested recursions. For these kinds of hardware descriptions, the compiler generates schematic netlists in which none of the nested recursion blocks are instantiated. In these cases, additional GDT instruction must be issued to instantiate recursively defined cells. However, it is possible to convert the recursion to one-level recursion that is functionally equivalent (illustrated in Example 4).

Example 4. One-Level Recursive Definition of an n Bit S/P Multiplier

A block diagram of the modified multiplier is shown in Figure 5.
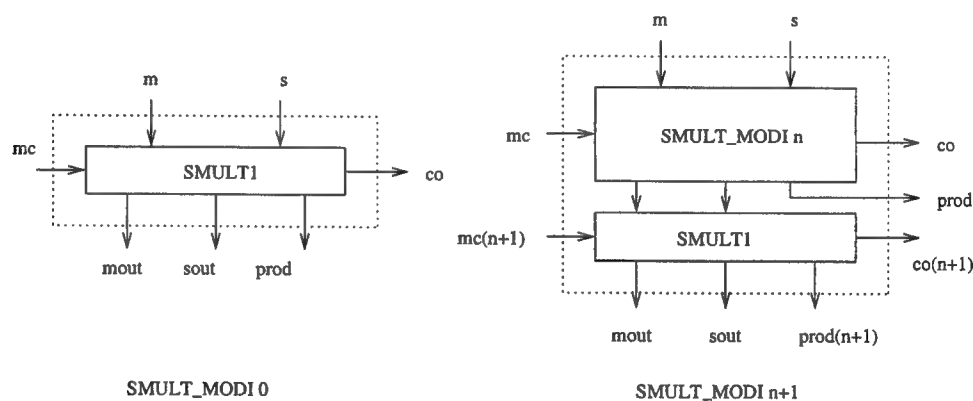
Figure 5: Diagram of the Modified Multiplier

SMULT_MODI is defined as functionally equivalent to *SMULTN* and uses only one level of recursion, thus it can be used to generate the layout automatically. The implementation description of *SMULT_MODI* in HOL is shown below:

8

```
SMULT1
|- ! mc m mout s sout prod co reset load clk.
      SMULT1(mc,m,mout,s,sout,prod,co,reset,load,clk) =
       WPROD1(mc,m,cout,prod,load,reset,clk) /\
       SADDER1(clk,prod,s,co,smout,reset)


SMULT_MODI
|- (! mc m mout s sout prod co reset load clk.
      SMULT_MODI 0 mc m mout s sout prod co reset load clk =
       SMULT1(mc 0,m,mout,s,sout,prod 0,co 0,reset,load,clk)) /\
    (! n mc m mout s sout prod co reset load clk.
      SMULT_MODI (n+1) mc m mout s sout prod co reset load clk =
      (? w1 w2.
       SMULT_MODI n mc m w1 w2 prod co reset load clk /\
       SMULT1(mc(n+1),w1,w2,sout,prod(n+1),co(n+1)reset,load,clk)))
```

The basic building block *SMULT1* does not include any recursive definition. It is only the combination of two basic building modules *WPROD1* and *SADDER1*, which are a one-bit partial product generator and a one-bit serial adder, respectively. Because the definition does not have any nested recursion, the layout can be generated automatically.

There is another type of recursive definition that should be avoided in the current HOL2GDT compiler, and this is illustrated in Example 5.

Example 5. Internal Lines in the Base Case Definition

Below is a HOL implementation description of an *n*-bit ALU.


```
nALU
|- (!H L ctl cin x y cout out.
      nALU 0 H L ctl cin x y cout out =
      (?yinv w1 w2 w3.
```

```
       inv  (ctl,y 0,yinv 0) /\
       FA   (x 0, yinv 0, cin, cout, w1 0) /\
       and2 (x 0, yinv 0, w2 0) /\
       or2  (x 0,yinv 0, w3 0) /\
       mux4 (H, L, w1 0, w2 0, w3 0, yinv 0, out 0))) /\
   (!n H L ctl cin x y cout out.
     nALU (SUC n) H L ctl cin x y cout out =
     (?yinv w1 w2 w3 cn.
       nALU n H L ctl cin x y cn out       /\
       inv (ctl,y (SUC n), yinv (SUC n)) /\
       FA  (x (SUC n), yinv (SUC n), cn, cout, w1 (SUC n)) /\
       and2(x (SUC n), yinv (SUC n), w2 (SUC n)) /\
       or2 (x (SUC n), yinv (SUC n), w3 (SUC n)) /\
       mux4(H, L, w1 (SUC n), w2 (SUC n), w3 (SUC n),
             yinv (SUC n),  out (SUC n))))
```

In the above case, the building block of the n-bit ALU consists of one inverter, a one-bit full adder, a two-input AND gate, a two-input OR gate, and one 4×1 multiplexer. The problem in this type of definition is that internal lines are used, and also indexed, in the base case definition. Unfortunately, this kind of definition cannot be handled by the HOL2GDT compiler and should be modified to include no internal lines in the base case definition. This modification can be accomplished by defining a one-bit ALU and using it as the base case definition. The corrected definition for the n-bit ALU is shown below.

```
ALU1
|- (!H L ctl cin x y cout out.
     ALU1(H, L, ctl, cin, x, y, cout, out) =
     (?yinv w1 w2 w3.
       inv  (ctl, y 0, yinv 0) /\
       FA   (x 0, yinv 0, cin, cout, w1 0) /\
       and2 (x 0, yinv 0, w2 0) /\
       or2  (x 0, yinv 0, w3 0) /\
       mux4 (H, L, w1 0, w2 0, w3 0, yinv 0, out 0)))
```

```
nALU
|- (!H L ctl cin x y cout out.
      nALU 0 H L ctl cin x y cout out =
         ALU1(H, L, ctl, cin, x 0, y 0, cout, out 0)) /\
    (!n H L ctl cin x y cout out.
      nALU (SUC n) H L ctl cin x y cout out =
      (? w.
         nALU n H L ctl cin x y w out /\
         ALU1(H, L, ctl, w, x (SUC n), y (SUC n), cout, out (SUC n))))
```

In the modified definition of the n-bit ALU, many details are hidden in the nALU definition, including all the internal lines from the base case definition as well as subcomponents in the base case and inductive case definitions.

2.3   Port Hiding

As mentioned briefly in Section 2.1, existentially quantified signals represent the internal connections between subcomponents, and these connections are usually invisible from outside the module. Another use of the existential quantifier is to hide some external ports to make the design simpler. For example, the block diagram of *SMULT_MODI* contains two ports that are not used outside the *SMULT_MODI* block, namely, the partial product *prod* and carry bits *co*. In case of multiplication, these signals are just for intermediate calculation, and there is no need to access the ports from outside the module. By using an existential quantifier, these two signals can be hidden.

Figure 6 shows the block diagram in which two external signals are hidden.

Comparing Figure 5 with Figure 6 reveals that the circuit becomes much simpler by hiding two unnecessary external signals. The HOL definition of *SMULTN* is shown below:

11
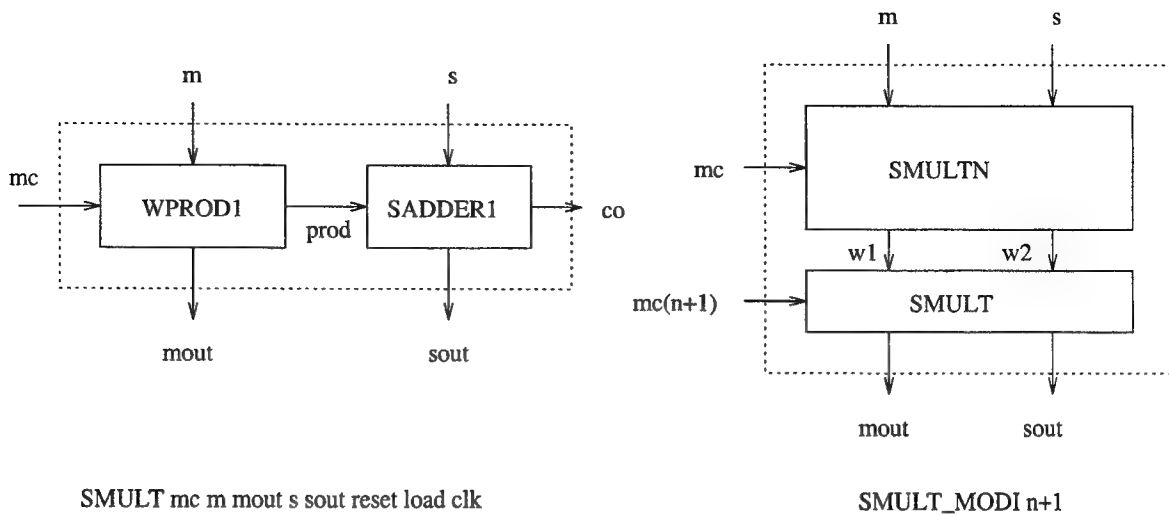
Figure 6: Diagram of Serial Multiplier with Two Hidden Ports

SMULT
|- ! mc m mout s sout reset load clk.
  SMULT(mc,m,mout,s,sout,reset,load,clk)=
    (? prod co. SMULT1(mc,m,mout,s,sout,prod,co,reset,load,clk))

SMULTN
|- (! mc m mout s sout reset load clk.
      SMULTN 0 mc m mout s sout reset load clk =
        SMULT(mc 0,m,mout,s,sout,reset,load,clk)) /\
    (! n mc m mout s sout reset load clk.
      SMULTN(n+1) mc m s sout reset load clk =
      (? w1 w2.
        SMULTN n mc m w1 s w2 reset load clk /\
        SMULT (mc(n+1),w1,mout,w2,sout,reset,load,clk)))

The definitions for *SMULT_MODI* and *SMULTN* have the same internal structures but different external structures in terms of input/output ports.

12

2.4 Input/Output Port Definition in HOL

One significant difference between HOL and the L language description is that there is no distinction made between input and output ports in HOL. However, in the L language description, every port must be designated as input, output, or both. Thus to translate the HOL definition to the L language program, this input/output port information must be provided in the HOL definition by the user.

Four HOL types are defined for this purpose: *variable*, *port_width*, *port*, and *module_port*. The syntax for the types is shown below.

```
variable : string
port_width = pin | bus num
port = in_port variable port_width |
        out_port variable port_width |
        inout_port variable port_width
module_port = module_name # (port)list
```

The port definition should be presented before the implementation description, and the variable names and their positions should be matched with those in the implementation description. Otherwise, the compiler will issue error messages. The port definition for the last serial multiplier description is shown below.

```
|- cell_port_def =
  ['SMULT1',
   [in_port 'mc' pin; in_port 'm' pin; out_port 'mout' pin;
    in_port 's'  pin; out_port 'sout' pin; inout_port 'prod' pin;
    out_port 'co' pin; in_port 'reset' pin; in_port 'load' pin;
    in_port 'clk' pin];
   'SMULT',
   [in_port 'mc' pin; in_port 'm' pin; out_port 'mout' pin;
    in_port 's' pin;  out_port 'sout' pin; in_port 'reset' pin;
    in_port 'load' pin; in_port 'clk' pin];
   'SMULTN',
   [in_port 'mc' (bus 1); in_port 'm' pin; out_port 'mout' pin;
    in_port 's' pin; out_port 'sout' pin; in_port 'reset' pin;
    in_port 'load' pin; in_port 'clk' pin]]
```

3 The GDT System

The GDT (Generator Design Technology) system is a commercial VLSI design environment for multiple levels of abstraction. That is, it can handle from the highest level (system) to the lowest level of abstraction (layout, mask) by integrating many different IC design tools such as integrated database, graphic editor, simulator, router, and design rule checkers, etc. Thus the GDT system provides an integrated environment for the design of cell-based ICs, as well as fully customized ICs. Since GDT is a large system a brief introduction will be helpful in understanding the HOL2GDT system.

3.1 Design Entry

The most important procedure in IC design is constructing a model for the design, followed by a simulation to verify its behavior. A higher level of modeling can be implemented with a box containing electrical terminals for the inputs and output signals. The behavioral model can be specified by programming languages. Complex, low-level models are realized by connecting functional logic components from component libraries. As the design steps progress, details are added to the model until enough information to specify the layout (mask) is gathered.

A fully customized or cell-based IC design should pass through many different design tools with different interfaces and operating styles, as well as independent databases. The translation between different databases wastes time and can introduce unexpected side effects. Errors also can be introduced each time a design is transferred between specialized tools. For example, a specific GDT layout cannot be translated into exactly the same layout in MAGIC because of the design rule difference between the two systems. Thus in conventional design tools, as a model passes through design procedures the productivity and the quality of the design deteriorate.

However, since GDT is a totally integrated system with all the required design tools and database included in the system, the possibility of the above problems can be minimized.

3.2 A GDT System Overview

The GDT system consists of six basic tools: the *L language*, *L database*, *Lsim* simulator, *Led* graphic editor, *M language*, and *cell library components*. A block diagram of the GDT environment is shown in Figure 7.

14
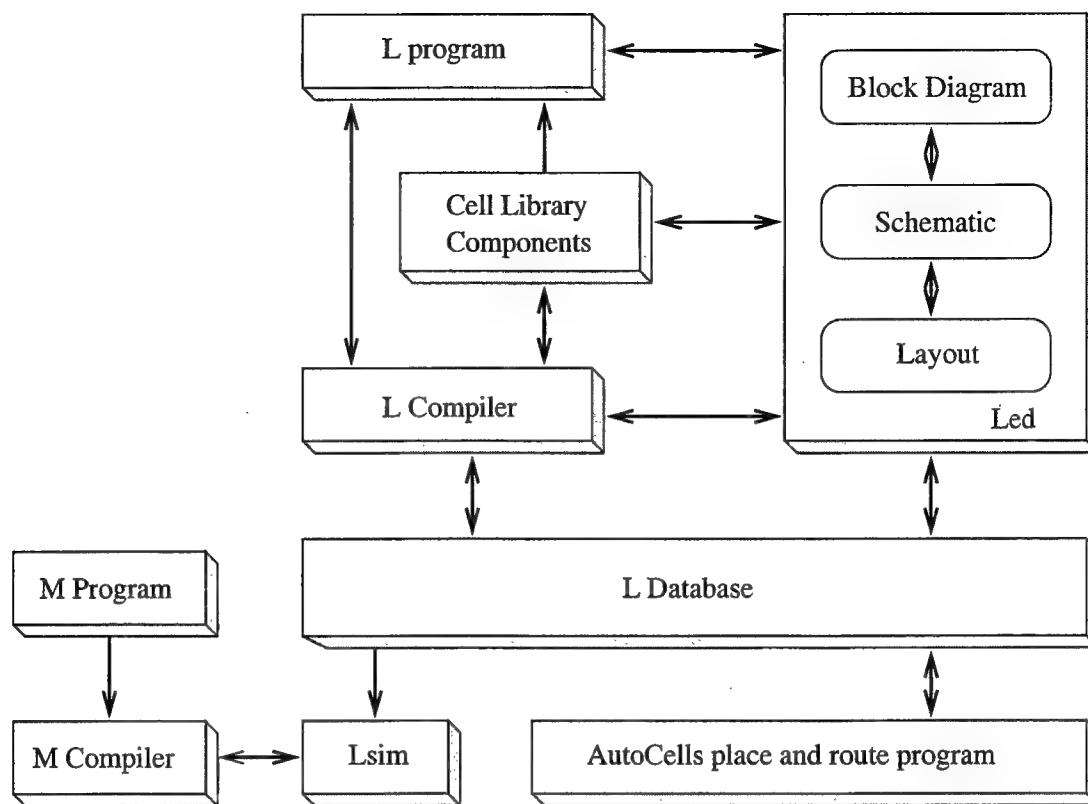
Figure 7: GDT System Environment

The design entry process progresses with the construction and validation of the design model through the automated tools provided in the GDT design environment.

3.3 GDT Human Interfaces

There are four user interfaces in GDT: the L compiler *Lc* and *L language*, L graphic editor *Led*, L database interface routines *Ldbi* and C function interface *Lx*, and the *GENIE procedural Language*. These interfaces enable users to access the L database and other L tools. With these interfaces the development and validation of the contents of the database can be achieved throughout the design process. However, the HOL2GDT system uses only *Lc* and *Led*, thus *Ldbi* and *Lx* will be not be discussed in this paper.

15

3.3.1 The L Language

A structural and functional description specifies how a circuit is formed in terms of the connectivity of its building blocks and what the circuit does. Silicon compilation is a translational process that generates a mask layout from the structural description, and this is sometimes called *module generation*. The L language is a procedural circuit layout language used to describe circuits by means of L programs. L programs, the source codes written in the L language, are called *module generators*. These L programs are compiled by the L compiler and beome part of the L database. More details and some actual L language examples will be covered in Section 4.

3.3.2 The L Compiler

The L compiler compiles L programs into design data procedures with geometric and electrical descriptions of a design. These design data procedures are stored in the L database. During the compilation of L programs, the L compiler can access to the L database to retrieve and update the information needed by L programs. It also has access to the utility programs, the so-called L tools that perform specialized functions such as routing, layout compaction, and placement. The L compiler consists of the following programs:

- Technology management program and instructions for adapting the L programs to new technology

- Symbolic icon generator for schematic capture of system-level architecture

- Behavioral model that simulates the icons used in schematic capture

- Layout generator that creates physical mask layout

- Test vector generator to provide Lsim simulator with a set of input signal sequences (vectors)

3.3.3 The L Graphics Editor (Led)

The Led graphics editor is an object-oriented graphical editor used to create icons, schematics, and actual layouts. It interacts directly with the L database. A schematic diagram of a design can be edited on the screen with the new graphic interpretation of the database displayed at the same time. In this way Led synchronizes the L

16

database and the graphical schematic view. As the L database supports three different views of a design (i.e., icon, schematic, and layout), Led can be used as an icon editor, schematic editor, and layout editor. It also creates simulation netlists for three different models. Finally, it handles standard cells and block routing. In summary, the Led offers the following facilities:

- Draw/edit hierachical schematics and layouts for any mask-defined technology

- Manipulate circuit objects

- Combine schematics and layouts

- Write out Lsim, AutoCells, AutoRoute, and SPICE netlists

- Verify a layout using the interactive rule checker

- Debug generator programs written in the L language

3.4 The L Database

The L database plays an essential role in the GDT design system. It captures the design and technology information required for developing and validating a design. The L database contains three types of information: *technology, geometric orientation,* and *netlist.*

3.4.1 Technology Information

The technology information database contains the information obtained from a technology file. This technology file defines all the primitive objects available within a technology and the design rule information that is specific to the manufacturing process being used.

3.4.2 Geometric Orientation Information

In a cell design, transistors, wires, and contacts are stored in the L database as primitive circuit elements. For each primitive the geometric information is stored in the L database for the purpose of physical construction, orientation of an object, and design-rule checking, compaction, and graphical editing.

17

3.4.3 Netlist Information

The L database contains netlist information of a cell—that is, information about the electrical connectivity of the cell. In addition, it contains information about simplifying the extraction of netlists and netlist-driven design-rule checking.

3.5 L Tools

The GDT system provides *L tools* for designing and validating full-customized IC and ASICs. L tools support complete placement, channel routing, compaction, design validation, and the creation of Lsim, SPICE and AutoCells netlists. In this paper the tools for placement and routing, design-rule checking, and creating an Lsim netlist are described.

3.5.1 Cell Placement and Routing Tools

GDT provides three kinds of place and routing tools. *Lroute* is a collection of automatic layout routing tools that perform block routing between instances and external terminals in order to build major blocks. The blocks and pads can be routed to perform chip assembly. *Lroute* also allows procedural routing within a specified section of a layout cell. *Explorer AutoCells* is an automatic placement and routing tool for laying out circuits using standard cells. Finally, *AutoRoute* performs block routing between instances and terminals.

3.5.2 Design Rule Checker Tools

Lrc is an object-oriented, hierarchical L language design rule checker. It performs design rule checking on L cells and provides interactive feedback about geometric and electric design rule violations. There are two modes in Lrc: *batch rule checking* is used for small designs and *selective rule checking* for large designs.

3.5.3 Lsim, Mixed-Signal, Multi-level Simulator

The *Lsim* simulator provides a general solution to design simulation. The system level, functional level, gate level, and switch level simulation modes can be mixed and simulated simultaneously. Lsim provides an infrastructure for many simulation algorithms. It also performs multi-level concurrent simulation on behavioral and structural models.

18

4 Schematic Description in the L language

In this section the L language, which is mentioned in Section 3.3.1, is decribed in detail due to its importance in understanding and using HOL2GDT methodology. This section is the summary of the GDT manual from Mentor Graphics. For more information or help, please refer to the Mentor Graphics L Language User Guide [7].

The L language is a procedural circuit-layout language that is used to describe circuits by means of programs called generators. It supports not only geometric primitives such as rectangles, polygons, and text, but also electrical circuit primitives such as transistors, wires, cantacts, cells with their geometries. It also supports the relative placement of all objects and contains routing statements for automatic wiring objects. In addition, it provides general-purpose programming language facilities sucsh as variables and control statements.

Writing a generator in the L language includes:

- Computation and flow control

- Transistors and wires that describe function and logic

- Geometry that describes physical layout

- Data that produce fabrication masks

- Hierarchy to manage the complexity of the design

The actual output from the L language is a description about how a chip is put together.

4.1 Key Concepts

There are several key concepts in the L language, as seen below:

- L files, the L language conventions, L keywords

- Names of objects in L

- Numerical variables and expressions

19

- Floating point numbers

- String variables and expression, string function

- Logical expressions

- Conditional statements

4.2   L Files

The basic unit in designing a module generator with the L language is a **cell**. An L program consists of one or more cells that are stored in one or more ordinary files.

The related syntax using L files is shown below.

```
<Ltool> [<options>] <L_file1, L_file2, ...>

where
   <Ltool>
           Lc         : The L language compiler
           Lrc        : The L language design rule checker
           Led        : L graphic editor
           AutoCells  : place and route program
```

4.2.1   L file structure

A typical L file structure is shown below.

```
L::TECH tech_name

Global variable declarations;
Arithmetic and string expressions;
Include other L files;
Generator calls;
```

20

```
CELL  a()
{
    local & exported variable declarations;
    generator calls;
    other : statements;
}

CELL  b()
{

    . . .

}
```

The first line, *L::*, specifies that this is an L file and prevents non-L files from being compiled. The second part, *tech_name*, specifies the technical file that contains all information for fabricating the actual chips. If the tech_name is not specified, it is assumed that any technology file may be used.

4.3   L language Conventions

There are several conventions concerning terminators, keywords, and comments for the L language. First, each statement should end with a *statement terminator* ";" as in the C programming language. There are two exceptions where the terminator is not needed: a starting line beginning with "L::", and such group statements as *CELL*, *WHILE*, and *IF_ELSE* that are enclosed with curly brackets. Keywords designated in the L language are all uppercase characters. Comment lines are denoted by pound symbols (£) and should end with a carriage return.

4.4   L language Keywords

There are two types of L keywords: technology-independent and technology-specific.

- Tech_independent keywords

```
. IF, ELSE, WHILE  : control statements
. GLOB             : global variable declaration
. CALL             : read an L file that contains a generator
. CELL             : cell declaration
. INST             : instance declaration
. WIRE             : wiring command for metal and poly lines
. FOPENR           : open file for reading
. LEFT             : position an object to the left of another
. UP               : direction for wire path
```

- Tech_specific keywords : defined in a technology file

```
. Layers
    MET, POLY, NDIFF, PDIFF, MET2, NWELL, PWELL, TN, TP, TD

. Contacts
    MPOLY, MNDIFF, MPDIFF, MNSUB, MPSUB, M1M2

. Terminals
    VDD, GND, IN, OUT, INOUT
```

4.5   Names of Objects in L

Except for polygons and rectangles, all objects in the L language have names.

4.5.1   Declaring an Object Name

Example:

```
TP    ptran    W = expression
```

The above statement means that a *p*-channel transistor is declared in a cell with a name *ptran* and channel width *W*. There are name buffers that can hold up to 128 characters, and user-defined names use lowercase characters. The names can start with $, _ . The brackets [ ] denote an index that can be attached only at the end of a name. However, the brackets cannot be used to index an array name. A grave accent (`) is used to construct names of objects with variable numerical indices. For example, the codes below generate 10 transistors with the names tran0, tran1,..., tran9.

```
NUM  i = 0;
WHILE (i < 10)  {
   TN  tran`(i)  AT (0, i*7);
   i++;
}
```

A period character (.) joins the name of an object with the name of one of its components. For example, when a cell has several terminals, one of the cell terminals can be accessed as shown below:

```
<cell_instance_name>.<terminal_name>
```

4.5.2   Scope of Names

Names are listed in three categories according to their visibility. *Global names* can be seen from anywhere in the L program. Technical data, numerical variables, string variables, and cells can be in the global name space. *Local names* are visible only within the cell where the names are declared. **NUM**, **INT**, string variables, transistor, contacts, terminals, instances of other cells, wires, and arrays of other cells can be in the local name space. A local name can be the same as a global name, and the local name supercedes the global name in a cell. Objects in the L database can be erased by the **DELETE** statement. *Exported names* are used to make local variables in a cell visible from other cells that are called from the cell containing the variables. An example of how a variable is exported using EXPORT function is shown below:

23

```
CELL abc{
   EXPORT NUM index = 55;
   NUM  i = 0;
   EXPORT i;

   CELL xyz()  {
      NUM i;
       . . .
   }
      . . .
}
```

There are two ways to declare exported variables. One way is to declare an exported variable using EXPORT and a type declaring keyword in one line. The other way is to declare a variable, and export the variable later using EXPORT. Whenever the values of i and *index* in the cell xyz() change, the cell abc() will receive new values. However, the changes of the exported variables occurring in the cell abc() do not affect the variables i and *index* in the cell xyz().

4.6 Numerical Variables

In the L language, the method of storing the values of variables in memory is somewhat different from that of other programming languages. That is, the integer and real numbers are stored in the L database in the same way, thus the keyword *NUM* is used to declare both real number variables and integer variables. To declare a variable as an integer, the keyword **INT** is used. The initialization is optional, however, as the default value is not defined and a variable must be assigned a value before being used. There is a scale factor that can determine the precision of the real numbers. A floating point number is multiplied by the scale, rounded to an integer, and then stored. For example, if a real variable s is declared and initiated using *NUM s = 48.35;*, then the internal data representation of the variable s in the L database becomes 4835.

24

4.7　String Variables and Expressions

String variables in the L language have the syntax seen below:

    STR <string_name> [= <string_expr>];

For example,

    STR isname = "This is an initial string";

is a typical string variable declaration.

4.8　String Functions

The L language offers various string functions as shown below:

- STRCAT(str1, str2): concatenates two strings
- FGETS(file_name)   : read a file into buffer. The file
                       must be opened before.
- GETS   : obtain a string from the standard
                       input (keyboard)
- NTOA(expr)   : converts a numerical expression into
                       an ASCII string
- STREQ(str1, str2) : if two strings are equal, gives 1,
                       else 0
- CELLNAME   : returns the current cell name
- GETDIR   : returns the current directory name
- GETFNAME   : returns the L file name being parsed
- GETLIB   : returns the name of library directory
- GETNEWFNAME   : returns a new file name which is unique
                       and writable for temporary data file
- GETNAME   : gives a unique name for an object
- GETTECH   : returns the technology name

25

```
e.g.

   STR  string;

   SPRINT(string, GETLIB," /", GETTECH, ".dir/icon");
```

In case the technology file name being used is "scmos", and the file is located at
scs/ind/tech, the string value stored in the *string* variable will be
   scs/ind/tech/scmos.dir/icon.

4.9   Logical Expressions

The L language supports almost the same logical expression as the C language. All
logical expressions yield 1 or 0 according to the result of the expressions, in which
1 means the evaluation result of the expression is true, and 0 means the result is
false. These logical expressions are used in conditional IF statements and WHILE
statements.

4.10   Conditional Control Statements

In the L language, there are two kinds of conditional control statements: *if_statements*,
and *while_statements*. The ablilities of decision and iteration come from these condi-
tional statements so that the module generators can be built.
   The syntax for the IF statement is shown below:

```
        IF (bool_expr)  {statements}


                or


        IF (bool_expr)  {
           statements1;
        }
        ELSE {
           statements2;
        }
```

For the first case, if the boolean expression evaluates to true, then the *statements* in the curly braces will be executed. If there is only one statement in statements1, the curly braces are not needed. For the second case, *statement1* will be executed if the boolean expression evaluates to true; otherwise, *statements2* will be executed.

To repeat a group of statements, the L language offers the WHILE statement, which can be used for the implementation of the recursive definitions. The syntax for the WHILE statement is show below:

```
WHILE (bool_expr)  {statements1}
```

The *bool_expr* is evaluated first. If it evaluates to true, then *statements1* will be executed, else it skips *statements1* and exits the WHILE loop. After executing statements1, it again evaluates the *bool_expr*. Thus the statements will be executed as long as *bool_expr* evaluates to true. Therefore, *statements1* must include a statement that allows exiting the loop; otherwise, the WHILE loop will never terminate. For example, consider the statement below.

```
NUM i = 1;
WHILE (i <= 10)  {
    TN   trans'(i);
    i++;
}
```

The above statements will generate ten of *n_channel* transistors named *trans[1]* to *trans[10]*. However, without the fourth statement, *i++*, the WHILE loop will generate *trans[1]* forever.

5 Using the L language

L programming is just a process of defining a cell. In the HOL2GDT system, a HOL description is converted to a schematic cell description in the L language. A schematic cell is represented in terms of the connectivity of the building blocks, and the building block is an instance of an existing cell. In this chapter the declarations of a schematic cell, input/output ports, an instance, and a net are explained in detail.

5.1 Schematic Cell Declaration

A schematic cell can be declared by using the format below:

```
SCHEMATIC  cell_name ([parameter lists])
{
    <group_of_statements>;
}
```

The keyword *SCHEMATIC* is used to declare a schematic cell, and *group_of_statements* can include one or more terminal, variable, cell_instance, net declarations, and conditional statements.

5.2 Input/Output Port Declaration

To communicate between cells, a schematic cell may have input/output ports. The I/O port is declared as a terminal and there are three types of terminal: *IN*, *OUT*, and *INOUT*. The syntax for terminal declaration is shown below:

```
terminal_type  name;
```

Here, the *name* can be an identifier or identifier with bracket.

28

5.3   Instance Declaration

An instance is an instantiation of a cell and it can be included as a component of cell definition. Before a cell is instantiated, it must be defined previously and if the cell definition is modified, then all the instances referring to it inherit the change. The syntax of instantiation is shown below.

```
INST  predefined_cell_name   inst_name;
```

The *inst_name* is the same name used in terminal declaration.

5.4   Net Declaration

A net describes how the components in a cell are connected to each other. The connections can be achieved only between legal L objects: *terminals* and *instance* terminals. There are two types of net in the L language: *WIRE* and *SIG*. *WIRE* is used to connect two terminals of blocks, and *SIG* is used to attach a signal to a terminal. That is, WIRE is used to connect IN/OUT ports to the component block terminals and SIG is used to connect block terminals. The syntax of net declaration is shown below.

```
WIRE   object  TO  object;
SIG    object  "signal_name";
```

The *object* is a terminal or instance_terminal name. Here the instance_terminal is a concatenation of an instance_name and a terminal_name.

5.5   Full Adder Example

An example of a full adder will clarify the mechanism of the cell definition. A full adder might be implemented by two half adders and one OR gate. The block diagram for a full adder is shown in Figure 8.

29
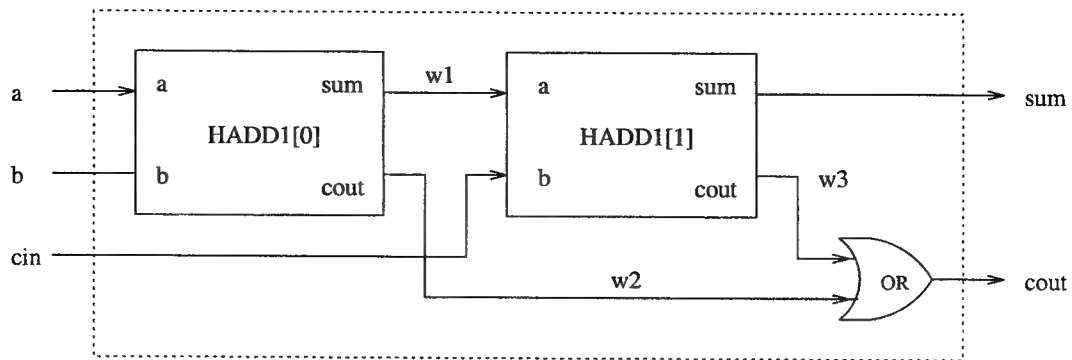
Figure 8: Diagram of Full Adder

It is assumed that the schematic cells for the half adder and the OR gate are already defined. The schematic cell definition of the full adder is shown below:

```
SCHEMATIC FADD1() {

    IN   a;
    IN   b;
    IN   cin;
    OUT  sum;
    OUT  cout;

    INST  HADD1  HADD1[0];
    INST  HADD1  HADD1[1];
    INST  or     or[0];

    WIRE  a    TO  HADD1[0].a;
    WIRE  b    TO  HADD1[0].b;
    WIRE  cin  TO  HADD1[1].a;
    WIRE  sum  TO  HADD1[1].sum;
    WIRE  cout TO  or[0].out;

    SIG  HADD1[0].sum   "w1";
    SIG  HADD1[0].a     "w1";
    SIG  HADD1[0].cout  "w2";
```

```
    SIG  or[0].in[1]   "w2";
    SIG  HADD1[1].cout "w3";
    SIG  or[0].in[0]   "w3";
}
```

At first the cell name of the full adder is defined as *FADD1()*, using the keyword *SCHEMATIC*. The *a, b, cin, sum* and *cout* are the I/O terminals of the FADD1() cell and are declared by using IN and OUT terminal types. Next, two half adders and one OR gate for the FADD1 cell are instantiated with a one-bit half adder *HADD1* and *or* cells that are defined previously. Next, the IN/OUT ports of the cell are connected to the terminals of cell component blocks, which can be done by using WIRE keywords. There are five IN/OUT terminals in a FADD1() cell, therefore the *WIRE* command needs to be used five times for the connections of all terminals. The terminal connections inside the FADD1() blocks are accomplished by the internal lines *w1, w2*, and *w3*. The L language uses **SIG** to declare the connection between a block terminal and a connection line. The terminals that have the same line name in a SIG declaration are to be connected each other. That is, the terminals *HADD1[0].sum* and *HADD1[1].a* are connected by the line labeled *w1*.

5.6 Conditional Control Statements

In the L language, the conditional control statements, *if_statement*, and *while_statement*, allow us to build module generators. That is, in case of a HOL recursive definition, the HOL2GDT compiler uses these conditional control statements to convert the recursive definitions into an L module generator. The syntax for the conditional control statements are covered in Section 4.10, and a detailed example is presented in the next chapter.

31

6 HOL to L Translation

This section covers the translation mechanism of the HOL2GDT system. This translation is accomplished by the structural mapping from HOL to L descriptions. First, *cell port definition* is covered and the transition mechanisms for HOL relational and recursive description will be explained later in detail.

6.1 Defining INPUT/OUTPUT ports in HOL

As mentioned earlier, no distinction is made for input and output ports in the HOL description. On the other hand, the direction of a port (input, output, or inout) must be specified explicitly in an L module description. This is the major difference between HOL and L schematic descriptions, and for the automation of the translation the directions (types) of the ports must be specified manually in the HOL description. Thus four HOL types: *variable*, *port_width*, *port*, and *module_port* are introduced for this purpose and shown below.

```
variable : string
port_width = pin | bus num
port =   in_port variable port_width
       | out_port variable port_width
       | inout_port variable port_width
module_port = module_name#(port)list
```

The *variable* is used to name the port. The *port_width* has two options: one is a *pin*, which is for a single line, and the other is a *bus* for multiple lines. There are three types of port directions: *in_port*, *out_port*, and *inout_port*. The *module_port* is used to represent the ports of a module and begins with *cell_port_def*. For example, the port definition and the relational definition of a full adder are shown below.

```
|- cell_port_def
   ['FADD1',
      [in_port 'a' pin; in_port 'b' pin; in_port 'cin' pin;
       out_port 'sum' pin; out_port 'cout' pin]]
```

```
|- ! a b cin sum cout.
     FADD1(a, b, cin, sum, cout) =
       (? w1 w2 w3.
          HADD1(a, b, w1, w2) /\
          HADD1(w1, cin, sum, w3) /\
          or(w3, w2, cout)
```

In the above definition the module name for the full adder is *FADD1*, and in the following port list there are three input ports, *a*, *b*, and *cin*, and two output ports, *sum*, and *cout*, all of which are pin types. The most important thing to remember is that the sequence of ports in the port definition must coincide with that in the relational definition.

6.2   Translating Relational Definitions

A typical relational definition in HOL can have the form below.

$$
\begin{aligned}
!\ in_1..in_n o_1..o_m.P(in_1, .., in_n, o_1, .., o_m) =& \\
?w_1..w_k.P_1(par\_list_1)\quad & \wedge \\
P_2(par\_list_2)\quad & \wedge \\
...\quad & \wedge \\
P_p(par\_list_p)\quad &
\end{aligned}
$$

P is the predicate being defined and *P1*, *P2*,... are the component blocks that are the instances of previously defined predicates. The input/output singal names are universally quantified before the predicate P, and P has these signals as its parameters. The predicate being defined must appear on the left side of the relational equation. On the right side of the equation the internal signals used to interconnect the subcomponents are existentially quantified. The instances of previously defined predicates (subcomponents of the predicate P) are conjoined with their parameter lists, *par_list1*, *par_list2*, and so on. The parameter lists of the instances can contain the universally quantified variables and the existentially quantified variables together.

The universally quantified variables are the external port names of the module P, and the existentially quantified variables are the line names used to interconnect the sub-components. The existentially quantified variables are hidden from the outside of module P. The relational definitions, coupled with the cell_port definitions, are translated into the L program in the sequences below.

1) The universally quantified variables of the predicate P (the predicate being defined) are declared as Input/Output terminals with terminal types IN, OUT, and INOUT by referring the cell port definition.
2) The first predicate P1 on the right hand side of a relation is renamed as a distinct instance of an existing cell definition. The distinct instance is generated by attching [num] to the end of each cell name where num is an integer.
3) The existentially quantified variables are declared as signals, and the variable names are enclosed with double quotation marks. However, this does not generate any L language code yet.
4) If a variable of the component P1 is universally quantified, then the corresponding instance terminal is connected to the corresponding I/O terminal by using the L constructor WIRE.
5) If a variable of the component P1 is existentially quantified, that is, if the variable has been declared as a signal, then the instance terminal of P1 is linked to the signal by using the L construct SIG. The instance terminal uses the port name defined in the cell_port definition for P1.
6) Repeat step 2) to step 5) for the remaining predicates, P2 .. Pp.

For example, consider a one-bit serial adder. A diagram of a one-bit serial adder is shown in Figure 9.

The one-bit serial adder is composed of one *d-register* and one *full adder*. The *cout* of the full adder is connected to the input port *in* of d-register. The cell port definition and the relational definition of the one-bit serial adder, *SADDER1*, are shown below:
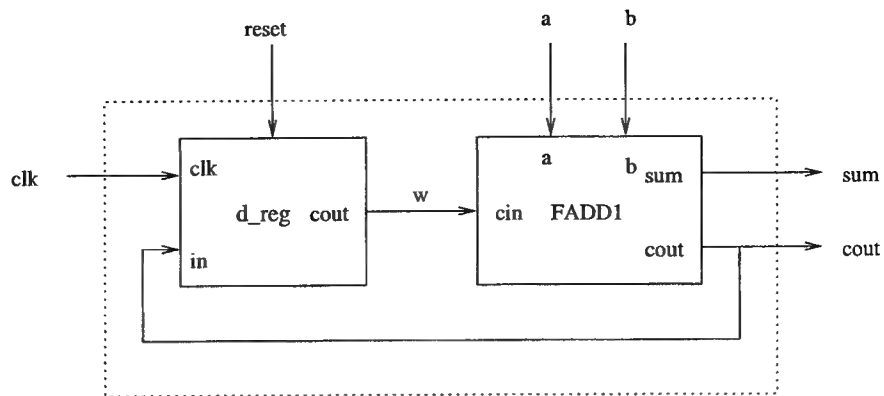
Figure 9: Diagram of One-Bit Serial Adder

```
|- cell_port_def =
   ['d_reg',
     [in_port 'in' pin; in_port 'clk' pin; in_port 'reset' pin;
      out_port 'cout' pin];
    'FADD1',
     [in_port 'a' pin; in_port 'b' pin; in_port 'cin' pin;
      out_port 'sum' pin; out_port 'cout' pin];
    'SADDER1',
     [in_port 'clk' pin; in_port 'a' pin; in_port 'b' pin;
      in_port 'reset' pin; out_port 'cout' pin;
      out_port 'sum' pin]]
```

SADDER1
```
|- ! clk a b reset cout sum. SADDER1 (clk, a, b, reset, cout, sum) =
     (? w. d_reg (cout, clk, reset, w) /\
           FADD1 (a, b, w, sum, cout))
```

The *FADD1* is defined in Section 6.1. The *d_reg* cell is a built-in macro-cell of GDT. It uses the basic GDT cell generator *msff*. The L language code of *d_reg* is shown below.

35

```
SCHEMATIC d_reg() {

    IN  in;
    IN  clk;
    IN  reset;
    OUT out;

    INST msff msff;
    WIRE msff.clk TO clk;
    WIRE msff.reset[0] TO reset;
    WIRE msff.reset[1] TO reset;
    WIRE msff.in  To in;
    WIRE msff.out To out;
}
```

The *msff* is a standard cell of GDT. The corresponding L file of the *SADDER1* is shown below.

```
SCHEMATIC  SADDER1()
{
    IN  clk;
    IN  a;
    IN  b;
    IN  reset;
    OUT cout;
    OUT sum;

    INST  d_reg  d_reg[0];
    WIRE  d_reg[0].in  TO  cout;
    WIRE  d_reg[0].clk TO  clk;
    WIRE  d_reg[0].reset TO reset;
    SIG   d_reg[0].out  "w";

    INST  FADD1  FADD1[0];
```

```
    WIRE  FADD1[0].a   TO  a;
    WIRE  FADD1[0].b   TO  b;
    SIG   FADD1[0].cin  "w";
    WIRE  FADD1[0].sum  TO sum;
    WIRE  FADD1[0].cout TO cout;
}
```

The translation mechanism is as follows: The input to the HOL2GDT compiler consists of the cell_port definition and the definition of *SADDER1*. The L schematic description is generated by scanning the definition of *SADDER1* from left to right. The HOL2GDT compiler finds the predicate name and produces the first line. **SCHEMATIC** is a reserved word in the L language, and every L program must start with this line except the comment lines. Then the compiler reads the external signals that are universally quantified. According to the information in the *cell_port definition*, HOL2GDT declares the variables. For instance, the first port in cell_port definition is *clk*, and it is defined as an input port. Therefore, the HOL2GDT compiler generates a code **IN clk;** and goes to the next parameter.

After the period mark the cell name with its parameter list follows. One thing to remember is that the cell name, the parameter names, and their positions in the definition should be matched with those defined in the *cell_port definition*. The HOL2GDT compiler checks this out and looks for the parameters which are existentially quantified. It marks these parameters as internal signals and remembers them.

Next, when the HOL2GDT compiler runs into a subcell name it instantiates it. For example, when it encounters the subcell name *d_reg*, it instantiates it using the **INST** construct. Thus an **INST d_reg d_reg[0]** line is added to the L schematic file.

After instantiating the subcell *d_reg*, the HOL2GDT compiler handles the parameters of d_reg: *cout, clk, reset,* and *w*. The first three parameters are universally quantified, thus they are to be connected with the external ports of the *SADDER1* cell. The HOL2GDT compiler generates three lines of net. For the terminal names of an instantiated cell, it uses the port names defined in the cell_port definition. Thus the first terminal name of the d_reg[0] is *in*, and it is connected to the first parameter of the subcell d_reg, *cout*.

The second port defined in d_reg cell_port definition is *clk*. Thus the instantiated terminal *d_reg[0].clk* is wired to the second parameter in d_reg subcell *clk*. The last

37

parameter w is somewhat different. It is existantially quantified and so it was stored as a signal previously. For a signal, HOL2GDT issues a **SIG** construct. Thus it generates a **SIG d_reg[0].out "w"**; line, and this line indicates that the instantiated cell terminal *d_reg[0].out* is connected to another instantiated terminal using the internal line w. Next, the FADD1 cell is instantiated and its parameters are processed. At the point of the third parameter w, the HOL2GDT compiler finds that it is a signal, thus it must issue the **SIG** instead of the WIRE construct. The third port of FADD1 is *cin*, thus the instantiated FADD1 cell terminal *FADD1[0].cin* is netted to the signal w. For a signal it generates two SIG lines in L description, and the instantiated cell terminals having the same signal names are to be connected to each other. For example, the *d_reg[0].out* terminal and the *FADD1[0].cin* terminal are connected with the line w.

6.3 Translating Recursive HOL Descriptions

This section explains how a recursive HOL description is translated to an iterative L program, that is, a parameterized module generator. A module generator is a cell description that accepts parameters as inputs, thus it generates cells that differ in size or in other characteristics according to the input parameter. Once a module generator is prepared, cells can be generated just by calling the module name with input paramters. This is convenient when there is a need to build arrays of identical cells.

Recursive definitions in the natural number domain consist of two predicates: one for the base case and the other for the recursive case. The predicate for the base case describes the primitive building block of a module. The predicate for the recursive case describes how this primitive building block should be connected to increase the size of the module by one. For example, an n-bit adder can be built out of n one-bit adders. The base case defines the one-bit adder itself, and the recursive case describes how a one-bit adder should be connected to build an n-bit adder.

The general recursive definition in HOL has the syntax below:

$$(!a_1..a_i.P_0a_1..a_i = Q(a_1, .., a_i))\wedge$$
$$(!na_1..a_i.P_{n+1}a_1..a_i =$$
$$?l_1...l_j.P_npar_list_1/ \ Q(par_list_2))$$

Here, n is the size of the cell to be built, and the variables $a_1..a_i$ represent the input

38

parameter list for predicates P and Q in the base case definition. The existentially quantified variables $l_1...l_j$ represent the signal lines for interconnection between sub-components; par_list_1 and par_list_2 are the parameter lists for cells P_n and B in the recursive case definition. They are equal to the input parameter list $i_1...i_k$ except that some of the parameters are replaced by the existentially quantified parameters $l_1..l_j$. The positions of $l_1..l_j$ in par_list_1 and par_list_2 determine how the basic building block is netted together. With the parameter lists and information on the position of the parameters, a recursive definition can be translated into a parameterized L module generator.

Since the translation mechanism for the recursive definition is somewhat intricate, the transition process is illustrated using a simple example of an n-bit ripple adder described in Section 2.2. For convenience, the block diagram of the n-bit ripple adder is shown again in Figure 10.
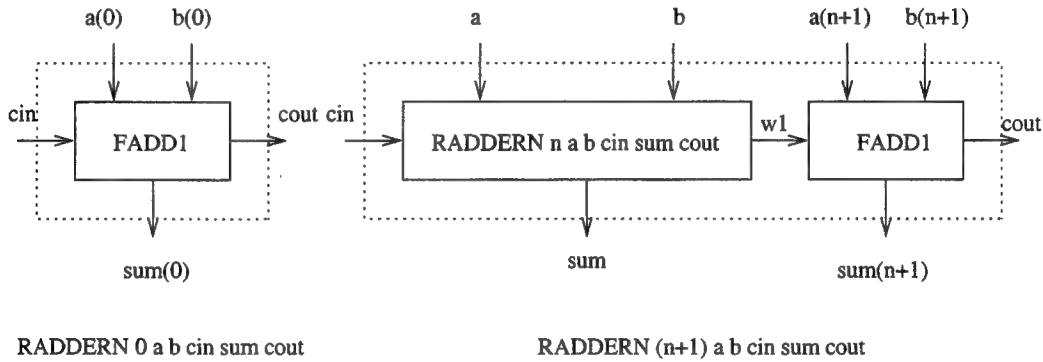
Figure 10: Diagram of an n-Bit Ripper Adder

The port definition of a one-bit full adder and n-bit ripple adder, along with the definition of the ripple adder, is shown below:

```
|- cell_port_def =
   ['FADD1',
      [in_port 'a' pin; in_port 'b' pin; in_port 'cin' pin;
       out_port 'sum' pin;  out_port 'cout' pin];
```

```
'RADDERN',
 [in_port 'a' (bus 1); in_port 'b' (bus 1); in_port 'cin' pin;
  out_port 'sum' (bus 1); out_port 'cout' pin]]

|- (! a b cin sum cout.
    RADDERN 0 a b cin sum cout = FADD1(a 0, b 0, cin, sum 0, cout))
                                   /\
   (! n a b cin sum cout.
    RADDERN (n+1) a b cin sum cout =
    (? w. RADDERN n a b cin sum w /\
          FADD1(a(n+1), b(n+1), w, sum(n+1), cout)))
```

In the base case of the *RADDERN* definition, zero is assigned to the cell size parameter. However, this does not mean that there is no instantiation of the base cell, but simply the instantiated cell number will start from 0. Therefore, if the cell size is n, there will be cell instances from 0 to $n-1$ in the translated L program.

The recursive part indicates that a ripple adder of size $n+1$ can be built with a ripple adder of size n and another one-bit full adder. That is, the recursive part defines the $n+1$ bit ripple adder with an n-bit ripple adder and a one-bit full adder.

The parameter lists and the position of the existentially quantified variables decide the interconnection mechanism. In the n-bit ripple adder case there is only one existentially quantified variable w, and it replaces two parameters: one for the output parameter of RADDERN *cout*, and the other for the input parameter of FADD1 *cin*. This indicates that the output port of RADDERN *cout* is to be connected to the input port of FADD1 *cin* using the internal line w.

In summary, the generation of the corresponding L module generator from a HOL recursive definition is based on the interpretation of the context in which a variable appears. The context is determined by two factors: *how the variable is quantified* and *where it locates*. Besides, a cell instance terminal name must be found according to the position of a variable in the predicate. To get this information, two abstract functions have been defined: *pos()*, and *port()*. With a predicate name P and its parameter a, the function application *pos(a,P)* returns the position of the parameter in the parameter list of predicate P. On the other hand, the function application of *port(n,P)* returns the n'th variable in the parameter list of predicate P.

The transition mechanism of a HOL recursive definition of an n-bit ripple adder to the L program is as follows:

The name of the predicate is *RADDERN*, and since it is defined recursively, the HOL2GDT compiler produces the first line.

```
SCHEMATIC RADDERN (INT hol_l_s=1)
```

SCHEMATIC is a keyword in the L language that specifies this is a schematic file. The name of the cell becomes *RADDERN* with default value for the cell size 1. This cell size can be set to a specific value after the HOL definition is translated into an L program. This will be covered in a later section. The HOL2GDT compiler uses the variable *hol_l_s* for the iteration limit parameter, which is declared to be an integer and initialized to 1.

Next, it refers to the cell port definition part of *RADDERN* and scans the port list from left to right. If the port has a type of *pin*, then it is declared to be a port variable by keyword **IN**, **OUT**, or **INOUT** according to its port type. Otherwise, if the port is not a pin type, then the HOL2GDT compiler remembers the port name and goes to the next port. For example, only *cin* and *cout* are of the pin type, and are defined as *in_port* and *out_port*, respectively. The compiler generates the two lines shown below:

```
IN   cin;
OUT cout;
```

The declaration for the rest of the parameters (which have the bus type) will be done during the iteration part, because every time a basic cell is instantiated it needs terminals for the interconnection between subcomponents or between the instance terminals and the external terminals.

Next, the compiler prepares for the iteration, but before the actual iteration it declares an iteration variable *hol_l_i* as an integer and initializes it to zero as seen below.

```
INT  hol_l_i;
hol_l_i = 0;
```

41

Then the iteration part follows as the HOL2GDT compiler produces the line below:

```
WHILE (hol_1_i < hol_1_s)  {
```

The limit for the iteration is given in the cell name definition as hol_l_s that is initialized to 1 by the compiler. Thus by default there will be one cycle of iteration, which means the size of the cell will be one.

The WHILE statement executes whatever statements in the loop as long as the boolean expression is evaluated to true. Since the variable hol_l_i is initialized as zero, the boolean expression results in true, and it performs the statements in the curly brackets.

In the iteration, the parameters with a bus type are declared first. Since there are three parameters having a bus type in the ripple adder example, the compiler generates the lines below:

```
IN    a[hol_1_i];
IN    b[hol_1_i];
OUT   sum[hol_1_i];
```

The compiler uses the iteration control variable, hol_l_i, to declare the parameter of the bus type and the parameters are indexed by this control variable.

After declaring all parameters of the bus type, the basic cell *FADD1* is instantiated and if it is the first cell instantiated (it checks the value of the variable hol_l_i to see if it is zero), then it searches the input variables defined outside the iternation and connects them to the corresponding terminals. In this case there is only one input variable defined *cin*, thus the compiler generates the lines below:

```
INST  FADD1  FADD1[hol_1_i];
IF (hol_1_i == 0) {
    WIRE FADD1[0].cin TO cin;
}
```

The above lines mean that the external input port *cin* of RADDERN is connected to the input port *cin* of the first instantiated full adder FADD1[0]. Here the HOL2GDT compiler uses two abstraction functions *pos()*, and *port()* to find the instance terminal name corresponding to *cin* port of RADDERN. The process of finding the instantiated terminal name for *cin* is as follows.

The input parameter *cin* indicates the outermost port, thus the predicate in the base case RADDERN is used to find the position of *cin* in FADD1. The function application *pos(cin, FADDERN)* gives the value of 3. To find the third port name in FADD1, the compiler uses another function application *port(3, FADD1)*, which gives the third port name in the FADD1 cell definition, *cin*.

The next step is to assign the port names for the instance terminals of bus type parameters. That is, the instance terminal for *FADD1[hol_1_i].a* is assigned to a port name *a[hol_1_i]*, and so on. In this case there are three parameters that have the bus type *a, b,* and *sum*. Thus the compiler produces the following three lines:

```
WIRE FADD1[hol_1_i].a    TO  a[hol_1_i];
WIRE FADD1[hol_1_i].b    TO  b[hol_1_i];
WIRE FADD1[hol_1_i].sum TO  sum[hol_1_i];
```

After netting the buses the compiler handles the existentially quantified parameter *w*. The compiler produces a line that asks whether the value of the iteration variable is not equal to zero. In case of zero (which means it is the beginning of the iteration, thus there is no other cell to be connected yet), the compiler skips this connecting procedure. If the value is not zero, then the compiler looks for predicates in which the existentially quantified parameter *w* appears. At first, the variable *w* appears in the RADDERN and is located in the fifth place. It can be found using *pos(w, RADDERN)*. The corresponding port name in the RADDERN port definition is *cout*. Again, the variable *w* appears in the predicate FADD1. The position value is 3, and the port name is *cin*. Thus the compiler knows that the instance terminal port of RADDERN *cout* is to be connected to the instance terminal port of FADD1 *cin*. One thing to remember is that the base building cell FADD1 is added to build RADDERN, thus the outermost port of RADDERN is that of the previously instantiated FADD1 cell. The code generated is:

43

```
IF (hol_1_i != 0)  {
    WIRE  FADD1[hol_1_i - 1].cout  TO  FADD1[hol_1_i].cin;
}
```

Next, the compiler generates a code that checks whether the value of the iteration variable *hol_l_i* is the same as that of the iteration limit variable *hol_l_s* minus one, which means that it asks if it is the last iteration procedure. If it is, the compiler generates a code that connects the last instantiated terminal to the external output port. In the above case, *cout* is the only output port that is pin type. The generated line is seen below.

```
IF (hol_1_i == hol_1_s - 1) {
    WIRE FADD1[hol_1_i].cout  TO  cout.
```

The final L program translated for the *n*-bit ripple adder is shown below.

```
SCHEMATIC RADDERN (INT hol_1_s=1)
{
    IN   cin;
    OUT  cout;
    INT  hol_1_i;
    hol_1_i = 0;
    WHILE (hol_1_i < hol_1_s) {
        IN  a[hol_1_i];
        IN  b[hol_1_i];
        OUT sum[hol_1_i];
        INST FADD1 FADD1[hol_1_i];
        IF(hol_1_i == 0) {
            WIRE FADD1[0].cout TO cin;
        }
        WIRE FADD1[hol_1_i].a to a[hol_1_i];
        WIRE FADD1[hol_1_i].b to b[hol_1_i];
        WIRE FADD1[hol_1_i].sum to sum[hol_1_i];
```

44

```
    IF(hol_1_i == 0) {
        WIRE FADD1[hol_1_i].cout TO FADD1[hol_1_i].cin
    }
    IF(hol_1_i == hol_1_s -1) {
        WIRE FADD1[hol_1_i].cout TO cout;
    }
    hol_1_i++;
}
}
```

Actually, the wiring mechanism in the HOL2GDT compiler follows the rules summarized in Figure 11.

	U / E	Var Occurence			iteration	Interconnection
		Bo	Pn	Pn		
case1	U	O	O		$i = 0$	WIRE B[0].t' TO x where pos(x, Pn) = k port(k, Po)=t pos(t,Bo) = k' port(k', B) = t'
case2	U	O	O	O	$0 \le i \le m$	WIRE B[i].t' TO x or x[i] where pos(x, Bo) = k pos(x, Bn)=k port(k, B) = t
case3	U	O		O	$i = m$	WIRE B[m].t TO x where pos(x, Bo) = k pos(x, Bn) = k port(k, B) = t
case4	E		O	O	$i \ne 0$	WIRE B[i-1].t' TO B[i].tt where pos(x, Pn) = k port(k, Po)=t pos(t,Bo) = k' port(k', B)=t' pos(x,Bn) = kk port(kk, B)=tt

Figure 11: HOL2GDT Compiler Translation Mechanism

45

Interestingly, all variables fall into the four cases and by matching the conditions of the variable the HOL2GDT compiler can issue the correct WIRE and SIG commands. Hence, the translation mechanism can be explained using the above rules. For convenience, some conventions are required, that is, P_0 is the predicate in the definition of the base case, and B_0 is the predicate for the building block in the base case. P_n is the predicate in the definition of the recursive case, and B_{n+1} is the building block in the recursive case. In the ripple adder case, B_0 corresponds to $FADD1_0$, P_n corresponds to $RADDERN_n$, and B_n to $FADD1_{n+1}$.

During the scanning procedure of each variable the compiler checks how the variable is quantified, in what predicate(s) it appears and, finally, the condition of the iteration. With this information, it finds out which case should apply. For example, the variable *cin* falls into case 1. The HOL2GDT complier knows that it should connect the input port variable *cin* somewhere. The variable *cin* is universally quantified, thus it is a primary input or output. By using the *pos()* and *port()* functions, the compiler can find that *cin* is an input port. Also, *cin* appears at the first iteration, that is, when $i = 0$. Therefore, the HOL2GDT compiler uses the first-case rule to find the instance terminal name to which *cin* should be connected by using the pos() and port() functions. The procedure of seeking the instance terminal is as follows. It first looks for the position of *cin* in the predicate RADDERNn by using the function application *pos(cin, RADDERNn)* and gives the value 3. Next, it uses the function application *port(3, RADDERN)*, which returns *cin*. With this variable *cin*, it finds the position of *cin* in $FADD1_0$, and this gives the value 3. Finally, the compiler uses the function application *port(3, FADD1)* to find the final instance terminal name, that is, *cin*. Thus the compiler issues **WIRE FADD1[0].cin TO cin**.

For the next three variables *a*, *b*, and *sum*, which are indexed by the recursion variable *n*, the compiler applies the second case. They appear in the base and recursive case definitions. Since they are universally quantified, they are primary inputs and outputs of the module. The compiler also uses the functions *pos()* and *port()* to trace the instance terminal names to be wired. Since these indexed variables appear in every iteration, they always occur in the same port locations. For example, the variable *a* appears in the first port of RADDERNn. Looking up the name of the first port of FADD1 by using the function application of *port(1, FADD1)*, which is *a*, the compiler can issue the WIRE command with the correct instance terminal name. This accounts for **WIRE FADD1[hol_1_i].a TO a[hol_1_i]**.

The variable *cout* falls into the third case. Since it is universally quantified, it is a primary input or output port. By checking the port definition of RADDERN, the compiler finds that *cin* is a primary output port, thus it must be used in the last

46

iteration to connect the output port from the instance terminal. The compiler checks the position of *cout* in $FADD1_{n+1}$, which is 5. With this value it looks for the fifth variable name in FADD1, which is also *cout*. Thus the compiler can issue the line **WIRE FADD1[hol_1_i].cout TO cout**.

For the *existentially quantified variables* the compiler applies the last case. Since the internal connections are not seen by the user, and they are not needed in the base building block cell, they will not occur as the parameters of $RADDERN_{n+1}$ or FADD1. The internal variables will occur only in $RADDERN_n$ and $FADD1_{n+1}$. The compiler finds the instance terminal name to which the internal variable should be wired as follows. First, it uses the function application *pos(w, RADDERNn)*, which returns 5. The fifth port in RADDERN is *cout*. Next, the compiler looks for the position of *cout* in $FADD1_0$, which is 5. Then it finds the fifth port variable of FADD1, which is *cout*. The internal variable w appears in $FADD1_{n+1}$. The position value of w in $FADD1_{n+1}$ is 3, and the third port variable in FADD1 is *cin*. Thus the compiler knows that the *cout* port of the previous stage should be connected to the *cin* port of the currently instantiated FADD1 cell, and issues the line below :

WIRE FADD1[hol_1_i - 1].cout TO FADD1[hol_1_i].cin.

In summary, to create an L module generator program from a recursive HOL description and a cell port definition list, the universally quantified parameters are declared as input or output ports, then the definition is scanned from left to right. The instance terminal names are determined and wired together using the rules in Figure 11. Notice that if the variable is a type of bus, it is indexed to be instantiated for every iteration.

47

7 n-Bit Serial Pipelined Multiplier Example

In this chapter a complete example of HOL2GDT methodology for designing a **five-stage pipelined bit serial multiplier** is covered. It is implemented using the HOL90.7 and GDT version 5.2.1. The entire procedures of the HOL2GDT system can be described with the design flow graph shown in Figure 12. The developing procedures of the multiplier are explained by following the sequences in Figure 12.
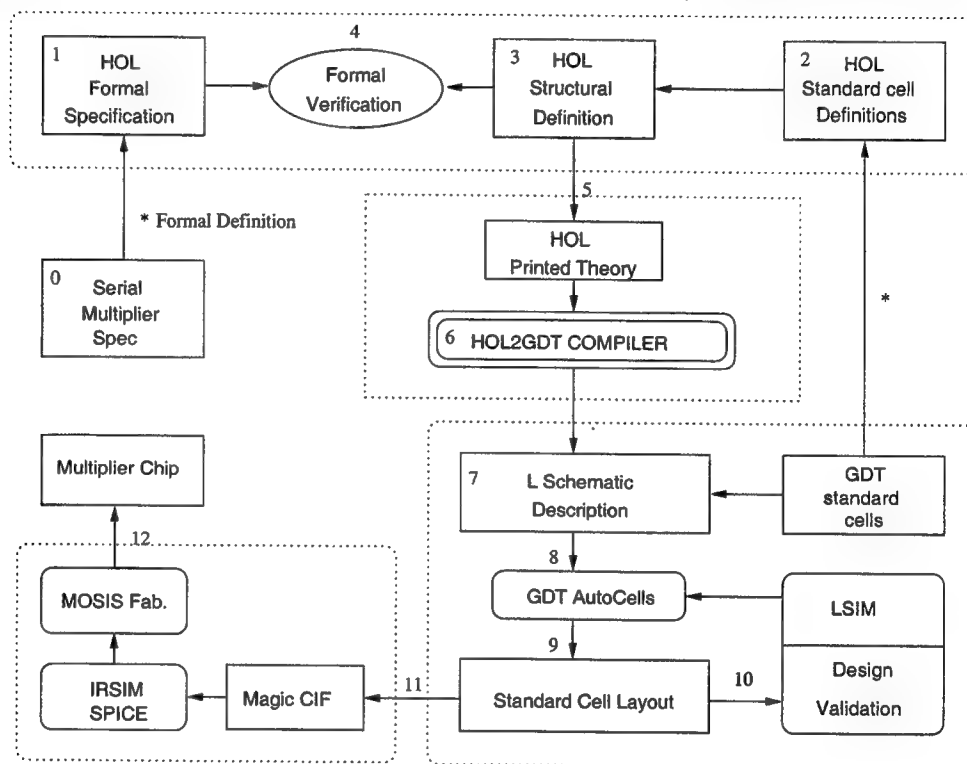
Figure 12: Flow Graph of HOL2GDT System

The multiplier design starts with a general multiplier specification (0). This specification is formalized using HOL notations (1). For the formalization of hardware systems, we have pre-defined standard cell definitions (2). With these definitions the structural(implementation) description of the multiplier is formalized (3). With both specification and implementation descriptions, we can perform a formal verification (4) using the HOL. After we get the correctness theorem of the multiplier design, we use the implementation description to get a HOL printed theory file (5). The hol2gdt

compiler (6) uses the printed theory to translates it into an L schematic description (7). Then the GDT auto-cell generator, *sc_build*, uses the schematic description to generate standard cells (8). With this standard cells, the L compiler generates a routing file (9). This routing file is used for the design validation through the *LSIM* simulator (10). The routing file is translated into a MAGIC CIF file for the embedding of pad frames (11). The MAGIC layout is tested using *IRSIM* or *SPICE* and sent to the MOSIS for the chip fabrication (12). Then we finally get a multiplier chip that is generated using formally verified definition. The contents of this chapter are as follows: In Section 7.1, the designing procedure of the multiplier building blocks is discussed in detail. In Section 7.2, we describe how building blocks can be defined using HOL notation. The detailed procedures of designing and verifying will be omitted because that is beyond the scope of this paper. Section 7.3 covers the translation procedure from the HOL to the L language. In Section 7.4, the layout generation procedures from the translated L schematic file to the final GDT circuit layout using GDT tools is explained, and the next chapter covers the testing and simulation procedures for the multiplier layout. The HOL implementation description and the translated L file for the multiplier are listed in the appendix.

7.1 Design Procedure of *n*-bit Serial Multiplier

The first chip fabricated using the HOL2GDT Linking System was the *pipelined serial multiplier*. In this section the design procedures of the multiplier from the basic component to the top level unit are described. However the main goal of this paper is not to describe the designing procedure of the multiplier itself, but to describe the design with HOL notation. Thus this section is just a preparation for the next section. The basic component of the multiplier is a full adder with reset input, *far*. The *far* acts like a full adder when the input to the reset port is **False**. With the reset input **True**, *far* still produces the *sum* result, but it makes the *cout* value **False**, regardless of the input values.

The *far* unit is implemented using basic gates such as two-input nand gates, inverters, three-input nand gates, two-input nor gates, and three-input nor gates. The block diagram of the *far* is shown in Figure 13.

The next step is to build a serial adder *sa* by combining a *far* and d-type register, *d_reg_nr*, which does not have the reset input. The serial adder *sa* keeps *cout*, the carry-out signal from the *far*, and uses it as the carry-input *cin* signal to itself for the next clock cycle. Therefore, there are no external *cin* and *cout* ports in the serial adder. The block diagram of the serial adder is shown in Figure 14.

49
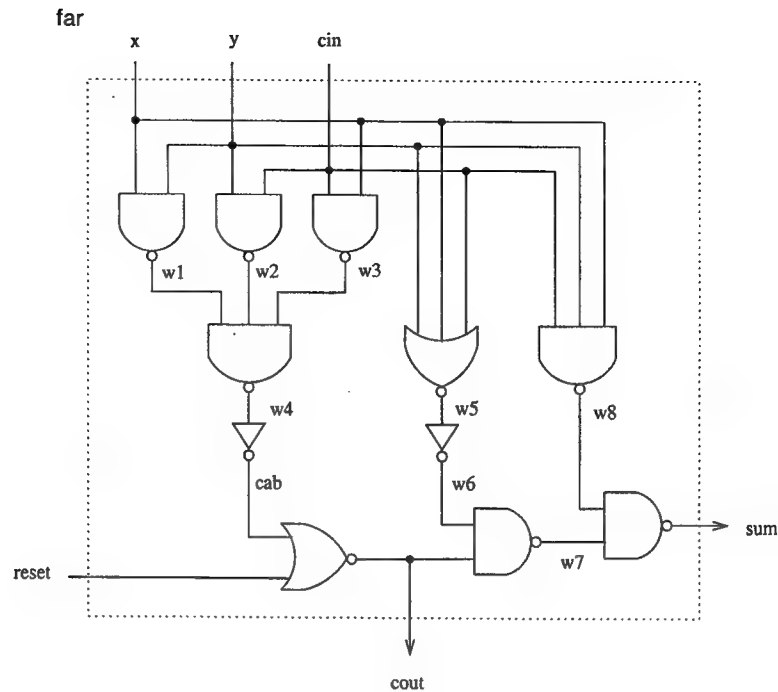
Figure 13: Block Diagram of a Full Adder with Reset

Now a component that can perform one-bit multiplication is required. Since multiplication is implemented by addition, the serial adder *sa* is used as a sub-component of the multiplier. In addition to the serial adder, three kinds of registers are needed: *d_reg_nr*, *d_reg_wqb*, and *wff*. The *d_reg_wqb* component has two output ports that always produce complementary outputs. It accepts the reset signal and delivers it to the output only when the clock signals change from high level to low level (trailing edge of clock signal). Otherwise, the output does not change at all. These registers are already pre-defined in the standard cells library of the GDT system. The *wff* component is just a d-type register with additional input signal drivers. The block diagram of *wff* is omitted here; however, the HOL definition of the *wff* is included in the appendix.

By using the components mentioned above, a one-bit multiplier cell, *cell*, can be built. The block diagram of the *cell* is shown in Figure 15.

There are six input and three output external ports in the *cell*. The input signals can be categorized into two parts: control signals and data input signals. The *clk* signal triggers the operation of the registers, and the *reset* signal clears the output of

50
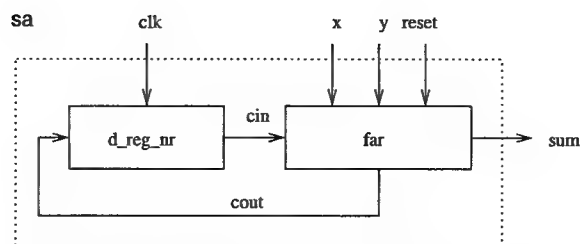
Figure 14: Block Diagram of a Serial Adder

the registers and initializes the carry input to the serial adder to zero. The x and y are data input ports.

For the output, *xout* is used to propagate the x input to the next stage. The *rout* port is also used to deliver the *reset* signal to the next stage. The *out* port is used to carry the result of the multiplication that is to be passed to the next stage.

The top level unit of the five-stage serial multiplier requires five one-bit multiplication units *cell* and four register *d_reg_nr* pairs between two adjacent *cells*. The block diagram of the five-stage serial multiplier is shown in Figure 16.

The block *bps* is just a serial connection of two inverters. The y signal is connected to the d-register through *bps*. A single *cell* and a *bps* block compose a *bcell*. A d-register pair is used to inter-connect two adjacent *bcells* and one *bcell* and d-register pair compose an *icell*. Thus to construct a five-stage, pipelined, bit-serial multiplier, four *icells* and one *bcell* are needed.

The next section explains how these building blocks can be defined with HOL notation.

7.2 HOL Implementation Description

In this section the procedure of describing the building blocks using HOL notation is explained.

7.2.1 Basic Gates Definition

In the previous section the basic gates such as *and*, *nor*, and even *inverter* gates are not designed, but taken for granted. However, there is no inherent library or built-in theory for the basic gates in HOL. Thus the basic gates must be defined before being used. To avoid building definitions of the basic gates in every implementation description, and also to conform to the terminology of the basic gates, the **gates.sml**

51
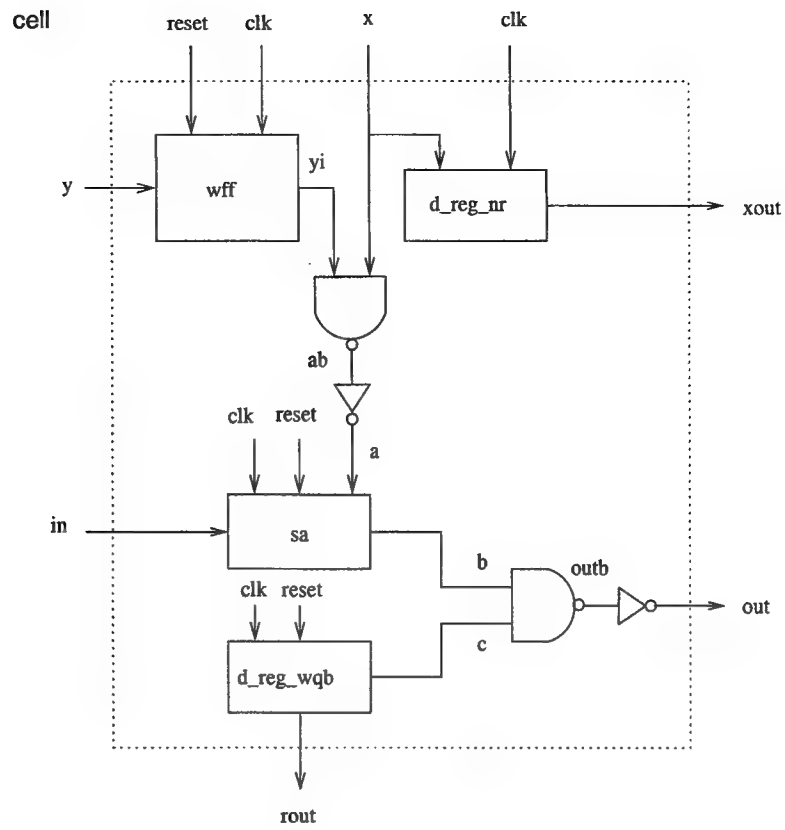
Figure 15: Block Diagram of a One-Bit Multiplier Cell

file that includes all the definitions of basic gates is provided. The gate names in the definition file are intended to be matched with the standard cell names of the GDT system. However, it is not always possible to do that for some gates. For example, the two input AND gate is named *and2*, because the word *and* is a reserved word in HOL90.7. The contents of the *gates.sml* file is listed in the appendix. To use the basic gates defined in the *gates.sml* file, the file must be loaded in a HOL session by executing the following procedures.

1. Copy the "gates.sml" file listed in the appendix to the working directory.
2. Execute below command.

pdm
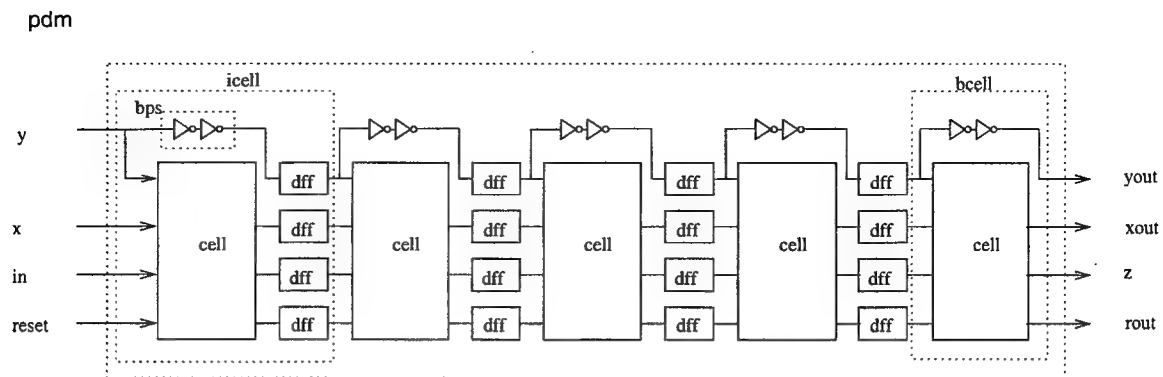
Figure 16: Block Diagram of Five-Stage Pipelined Serial Multiplier

```
        %ho190 < gates.sml          (* % is a prompt *)
```

3. In the HOL implementation description file, include two commands
    after new_theory command line.

    ```
    new_theory "pdm";   (* pdm is the theory name *)

    new_parent "gates";
    add_theory_to_sml "gates";
    ```

The second step generates the theorem file for the gates.sml definition file. The
theorem file is needed to execute the commands in step 3.

7.2.2 Defining Noniterative Structure Components

This section describes how the noniterative structured components are defined using
HOL notation. First, the *far* component in Figure 13 consists of two inverters, five
two-input nand gates, one two-input nor gate, and one three-input nor gate. All
of these gates are defined in the *gates.sml* file. It does not include any iterative
structures (repetition of the same components), so it should be represented with
relational description (refer to Section 2.1). The HOL definition of the *far* component
is shown below:

53

```
val far = new_definition
    ("far",
    (--'! x y cin reset sum cout. far (x, y, cin, reset, sum, cout) =
          ? w1 w2 w3 w4 w5 w6 w7 w8 cab.
                nand(x, y, w1) /\
                nand(x, cin, w2) /\
                nand(y, cin, w3) /\
                nand3(w1, w2, w3, w4) /\
                inv(w4, cab) /\
                nor(cab, reset, cout) /\
                nor3(x, y, cin, w5) /\
                inv(w5, w6) /\
                nand(cab, w6, w7) /\
                nand3(x, y, cin, w8) /\
                nand(w7, w8, sum)'--));
```

There are six external ports in the *far* block. These external ports must be universally quantified by the ! symbol. Keep in mind that there are no commas between the external port names. After the period mark, the cell name (here, *far*) is followed. Then all of the external signals should be included in the parentheses as the parameters of the predicate *far*. Here, commas are needed between parameters.

Next, the internal signals that act as connection lines between components should be existentially quantified. Then the subcomponents of *far* are listed with their parameters. Between the subcomponents a conjunction mark ∧ is placed. The subcomponents that have a common internal signal name in their parameter lists are to be connected. For example, the output port of the first *nand* gate is to be connected to one of the input ports of the first *nand3* gate by the line labeled *w1*. If there is no error in the definition (usually typographical errors or using a definition name that has been used previously), HOL produces the following theorem.

```
val far =
  |- !x y cin reset sum cout.
        far (x, y, cin, reset, sum, cout) =
        (?w1 w2 w3 w4 w5 w6 w7 w8 cab.
           nand (x, y, w1) /\
```

```
nand (x, cin, w2) /\
nand (y, cin, w3) /\
nand3 (w1, w2, w3, w4) /\
inv (w4, cab) /\
nor (cab, reset, cout) /\
nor3 (x, y, cin, w5) /\
inv (w5, w6) /\
nand (cab, w6, w7) /\
nand3 (x, y, cin, w8) /\
nand (w7, w8, sum)) : thm
```

Using the above method, other basic components that do not have iterative structures *sa*, *wff*, *cell*, *bcell*, and *icell* can be defined.

7.2.3 Defining Iterative Structure Components

This section describes how the iterative structure is defined with HOL notation. The top level description of the five-stage pipelined serial multiplier *pdm* consists of two parts: the first four stages, *ipdm*, and the last stage, *bcell* of the multiplier (refer to Figure 16). Since no registers for the pipelining are needed in the final stage, *bcell* is used instead of *icell*.

The *ipdm* is built by cascading (iterating) four *icells*, thus it can be defined in a recursive way (refer to Section 2.2). The HOL definition for the *ipdm* (for general n stages) is shown below.

```
val ipdm = new_definition
    ("ipdm",
    (--'(! x y in1 reset clk yout xout z rout.
            ipdm 0 x y in1 reset clk yout xout z rout =
            icell (x, y, in1, reset, clk, yout, xout, z, rout))  /\

        (!n x y in1 reset clk yout xout z rout.
            ipdm (SUC n) x y in1 reset clk yout xout z rout =
            (? xi yi zi ri.
                ipdm n x y in1 reset clk yi xi zi ri  /\
                icell (xi, yi, zi, ri, clk, yout, xout, z, rout)))'--));
```

The recursive definition of *ipdm* consists of two parts: the base case definition and the inductive case definition. The base case describes the primary building block of the *ipdm*, and the inductive case describes how the additional building block should be connected to increase the size of the *ipdm*. The base case definition indicates that the primary component of the *ipdm* is *icell*, and the inductive case definition reveals the output ports of the *ipdm*: *yout*, *xout*, *z*, and *rout* should be connected to the input ports of the attaching *icell*; and *x*, *y*, *z*, and *in* through the internal lines *yi*, *xi*, *zi*, and *ri*, respectively. Finally, the top-level definition *pdm* is acquired just by combining the *ipdm* and *bcell*.

7.2.4   Adding Input/Output Port Definition

As mentioned in Section 2.4, one more piece of information, the *input/output port definition*, is needed to translate the HOL definitions into the L language program. In HOL implementation description there is no distinction for the input and the output ports. However, the ports must be declared as input, output, or for both in the L program. This information should be supplied by attaching the *cell port definition* to the block definition in the HOL description. For example, the cell port definitions for the first component *far* and top level component *pdm* are shown below:

56

```
var cell_port_def1 = new_definition
    ("cell_port_def1",
    (--'cell_port_def1 =
        ["far",
           [in_port    "x"       pin;
            in_port    "y"       pin;
            in_port    "cin"     pin;
            in_port    "reset"  pin;
            out_port   "sum"    pin;
            out_port   "cout"   pin]]'--));

var cell_port_def8 = new_definition
    ("cell_port_def8"
    (--'cell_port_def8 =
        ["pdm",
           [cell_sixe   "n";
            in_port    "x"       pin;
            in_port    "y"       pin;
            in_port    "in1"     pin;
            in_port    "reset"  pin;
            in_port    "clk"     pin;
            out_port  "yout"   pin;
            out_port  "xout"   pin;
            out_port  "z"       pin;
            out_port  "rout"    pin]]'--));
```

After adding the cell port definitions, the file should be closed using the two lines below:

```
close_theory();
export_theory();
```

Executing the *close_theory()* finishes a session in draft mode and switches the system to proof mode. The changes made to the current theory segments are writ-

ten to the theory file associated with it. If the HOL session ends without invoking *close_theory()*, the modifications made to the session do not persist to future HOL sessions. After the theory is closed, the theory segment may be extended by using the *extend_theory()* command.

Executing the *export_theory()* exports the theory to the disk, thus generating a theorem file, *pdm.thms*, in the directory where HOL is invoked.

From now on, this implementation description of the multiplier can be used for the formal verification. The correctness theorem that shows the implementation description of the multiplier implies or is equal to the specification description of the multiplier. Since the formal verification procedure usually requires quite a long time, it may be postponed. However, the verification must be done before the layout is sent to the MOSIS for fabrication.

In the *sml* file, comments may be included to help the interpretation of user. However, these are not needed in the HOL2GDT compiler, because they may add complexity to programming the HOL2GDT compiler. Thus instead of the *sml* file, the *printed theory* file is used for the input to the HOL2GDT compiler. The printed theory file can be generated by the following steps.

At first, execute command below in HOL session.

```
print_theory "pdm";
```

The string, *pdm* is the theory name that was declared by the *new_theory* command at the beginning of the theory file. The *print_theory* command prints the theory name, parents, type constants, term constants, axioms, definitions, and theorems defined in the *pdm* theory. To save the printed theory into a new file, the new file should be opened with the name *pdm.print* and then the printed theory should be copied by clipping and pasting. However, the *printed theory file* may not yet be used as the input to the HOL2GDT compiler, because some changes must be made to the printed theory file. These may include collecting all cell port definitions and combining them into one definition and placing it at the beginning of the file to make the programming of the compiler easier. The *ipmap* program was developed to implement this.

The syntax for the ipmap is below:

```
ipmap <input file> <output file>
```

Here the input file name can be *pdm.print* and the output file name can be *pdm.*
There is one constraint in designating the output file name. That is, it may not
include any dot(.) extension name. The reason for this constraint is explained in the
next section. The final form of the HOL implementation description that is ready to
be used as an input to the HOL2GDT compiler is shown in the appendix.

7.3   Translating to the L program

In this section the procedure for translating the HOL implementation file into an L
language program is explained. The output from the *ipmap* program is used for the
input to the HOL2GDT compiler. As mentioned in the previous section, the input
file name to the HOL2GDT compiler may not have a dot extension. This constraint
comes from the syntax of GDT commands that are to be used from now on. The
syntax of the HOL2GDT compiler is below:

```
HOL2GDT  -o <output_file.S>  <input_file>
```

Attaching the suffix, .S, to the output file name is mandatory. In later procedures,
a file name with the .S suffix is required. The actual command for translating the
pdm file to an L program file is as shown below:

```
HOL2GDT  -o pdm.S  pdm
```

The compiler generates an L schematic file *pdm.S* and also two additional files,
pdm.def and *pdm.macro.* These two files include the parameter values for the pa-
rameterized GDT standard cell generators and the L language definition of macro
cells supported by the HOL2GDT compiler. GDT's placement and routing tools will
utilize these standard cell parameters to create necessary standard cells, and these
standard cells and macro cells are then used in building the multiplier layout.

7.3.1 Standard Cell Generator Definitions

The GDT tool suite provides a set of parameterized standard cell generator functions. These generators are supplied with a set of input values that denote the parameters to the generator function. For example, let *nand* be a standard cell generator function. The function is instantiated by an L language command, **CALL**. Then, the *nand* generator function is used to create a two-input nand gate. Each function has to be given a name label, such as *nand2*. Each function being instantiated must be supplied with a set of input parameter values. Thus to create a 2-input nand gate called *nand2*, the L language command line will be as below.

```
CALL nand CELL nand2 (2,1,1,0);

        where nand    - standard cell generator function being called.
              nand2   - instance name of the standard cell.
              2       - number of inputs.
              1       - number of outputs.
              1       - number of input drivers.
              0       - number of output drivers.
```

A more detailed description of input parameter syntax and semantics can be found in the standard cell manual [4]. The HOL2GDT compiler currently uses 11 standard cell generator functions to create and support a library of 27 standard cells. A typical *pdm.def* file is shown below:

```
L:: TECH ANY
{
# And gate
CALL sc_and CELL and (2,1,1,2,1,1,0);  # 2-input
CALL sc_and CELL and3 (3,1,1,2,1,1,0); # 3-input
CALL sc_and CELL and4 (4,1,1,2,1,1,0); # 4-input
CALL sc_and CELL and5 (5,1,1,2,1,1,0); # 5-input
# Or gate
CALL sc_or CELL or (2,1,1,1,1,1,0);   # 2-input
CALL sc_or CELL or3 (3,1,1,1,1,1,0);  # 3-input
CALL sc_or CELL or4 (4,1,1,1,1,1,0);  # 4-input
```

```
CALL sc_or CELL or5 (5,1,1,1,1,1,0); # 5-input
# Nand gate
CALL sc_nand CELL nand (2,1,1,0);  # 2-input
CALL sc_nand CELL nand3 (3,1,1,0); # 3-input
CALL sc_nand CELL nand4 (4,1,1,0); # 4-input
CALL sc_nand CELL nand5 (5,1,1,0); # 5-input
# Nor gate
CALL sc_nor CELL nor (2,1,1,0);     # 2-input
CALL sc_nor CELL nor3 (3,1,1,0);    # 3-input
CALL sc_nor CELL nor4 (4,1,1,0);    # 4-input
CALL sc_nor CELL nor5 (5,1,1,0,0); # 5-input
# Xor gate
CALL sc_xor CELL xor (1,1,0);
# Xnor gate
CALL sc_xnor CELL xnor (1,1,0);
# Inverter gate
CALL sc_inv CELL inv (1,1,1,0);
# Master-slave flip flop
CALL sc_msff CELL msff_wrqb (1,1,1,0); # with reset and Qb output
CALL sc_msff CELL msff_nr (0,1,1,0); # no reset
CALL sc_msff CELL msff (3,1,1,0); # vanilla master-slave
# Static latch
CALL sc_latchs CELL lsl_wr (1,1,1,1,1,0); # with reset
CALL sc_latchs CELL lsl_nr (0,1,1,1,1,0); # no reset
CALL sc_latchs CELL latchs (1,1,1,1,1,0); # vanilla latch
CALL sc_latchdd CELL latchdd (1,1,1,1,1,0); # dynamic latch
# Inverted Tristate buffer
CALL sc_tbfi CELL tbfi (0,1,1,0);
}
```

The standard cell library supported by HOL2GDT is open-ended and can be further extended to enhance the design capabilities as required. The standard cell library has one-to-one correspondence to the components in the *gates.sml* file described earlier in Section 7.2.1. The *gates.sml* file describes behavioral definitions of the standard cells in HOL. The HOL2GDT compiler uses these definitions in *gates.sml* to create a *pdm.def* file that contains the standard cell generator function calls and appropriate input parameter values for the generator functions.

61

7.3.2 Macro Cell Definitions

Macro cells are commonly used in any standard cell-based design methodology. Macro cells are constructed using one or more existing standard cells to achieve a frequently used functionality in hardware. The HOL2GDT compiler has a macro cell library that currently consists of 12 macro cells as listed below.

```
D_REG       - D register using master-slave flip flop
D_REG_NR    - D register with no reset
D_REG_WQB   - D register with inverted Q output
D_REG_WRQB  - D register with reset and inverted Q output
BUFFER      - Buffer using dynamic latch
BUFFER_S    - Buffer using static latch
VDD         - VDD power connection, used for tied high ports.
GND         - Ground connection, used for tied low ports.
BY_PASS     - Wire pass through using two cascaded inverters.
PASS        - Pass transistor.
BUS2WIRE    - Split a bus into wires.
WIRE2BUS    - Group wires to form a bus.
```

The macro cells are described using an L language program. The HOL2GDT macro cell library maintains a list of macro cells currently supported by the compiler. The compiler reads this list at run time from the library and writes the L language programs for each supported macro cell to the *pdm.macro* file. A typical macro definition file is listed in Appendix B.

A design is fully described by means of three design files, *pdm.S*, *pdm.def*, and *pdm.macro*. Once the pdm.S file is generated there is one thing to do with the pdm.S file manually. That is, if the HOL definition file includes one or more recursive definitions, then the values for the iteration variables in the translated L schematic program must be assigned (please refer to Section 6.3). By default the HOL2GDT compiler assigns 1 for each parameter n in the recursive definition. Thus if no adjustment is made in the *pdm.S* file, then a one-stage multiplier will be generated. To get the desired size of multiplier, the value of the *hol_Ls* parameter in the *pdm.S* file should be changed. For the five-stage, pipelined, serial multiplier, five *icells* and four pipeline stages are needed. This can be implemented by assigning the value 4 to the *hol_Ls* parameter in the *ipdm* cell generator.

62

7.4 Generating the Actual Layout Using CAD Tools

From now on all the procedures are implemented in VLSI CAD tool environments. The flow of the procedures is shown in Figure 17.
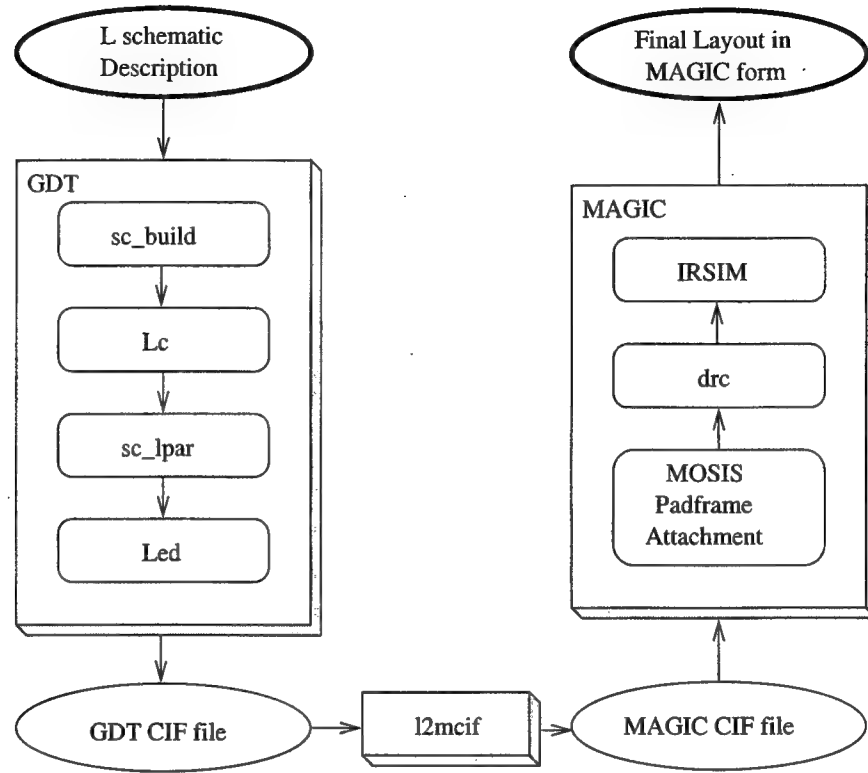
Figure 17: Procedure of Generating the Actual Layout

The above figure shows the procedures to get a MAGIC layout from an L schematic description. HOL2GDT uses two VLSI tools, GDT and MAGIC. The L schematic description is input into the GDT system and the system builds standard cells, does placement-and-routing, and converts the layout into CIF format. Since the CIF format supported by the GDT system differs from that of MAGIC, the GDT CIF file is translated into MAGIC CIF format using the *l2mcif* program. With the MAGIC CIF file, the MAGIC layout is retreived, pad frames are embedded into the design, and the design is tested using the design rule checker *drc* and the switching level simulator *IRSIM*. Details for the above procedures are covered in following sections.

7.4.1 Building Standard Cells

The first thing to do in the GDT system is to generate standard cells required for building the multiplier circuit layout. The *sc_build* is one of the supporting programs in the GDT system that refers to the standard cell library. It is used to build a set of standard cells. The syntax to build a set of standard cells is:

```
sc_build -t <tech> <options>
```

The options are

```
-l lib                Specifies the technology library.
-L lcell_lib          Specifies the Lcompilers library.
-T lcell_tech         Specifies the Lcompilers technology category.
-V view               Specifies the VIEW of the cells to be created.
-x                    Specifies that the output file should be in
                      binary format.  The default format is ASCII.
-o output_file        Specifies the name of the Lc output file.  The
                      default output name is stdout.
-e error_file         Specifies the name of the Lc error file. The
                      default file name is Lerror.
-g cell_control_file  Specifies the name of the standard cell
                      control file.  The default name is sc_ctl.
                      If the file sc_ctl is not found in the current
                      directory, default values will be used.
cell_definition_file  Specifies the name of the standard cell
                      definition file.  The default name is sc_def.
                      If the file is not found in the current
                      directory, a default cell definition file will
                      be used.
```

Thus the actual Unix shell command line to generate the standard cells for the multipilier is as shown below.

```
sc_build  -x -t scmos  -V SCHEMATIC -g gdt.ctl -o pdm.X  pdm.def
```

The above command means that the output file is in binary format, the technology
file to be used is *scmos*, the SCHEMATIC view will be generated, the cell control
file name is *gdt.ctl*, the output file name is *pdm.X*, and the cell definition file name
is *pdm.def*. The cell control file *gdt.ctl* must be in the working directory. For more
detailed information, please refer to the *L compiler* user guide [7].

7.4.2 Generating a Routing File

The next procedure is to generate the routing file using the L language compiler *Lc*,
which is a utility program for translating L programs into other netlist and geometric
formats. The command we used for generating the routing file is shown below:

```
Lc -t scmos -inc pdm.X  -inc pdm.macro  -o pdm.R  -AutoCells pdm.S
```

The above command means the technology file is *scmos*, which checks whether
the pdm.X and pdm.macro files are included. The output file name is *pdm.R*. For the
netlist information it uses the *pdm.S* file. The output file is *pdm.R* and some controls
can be applied manually on this file by using *TERMPLACE* commands at the end of
the *IF (_LPAR_NET)* clause.

```
TERMPLACE  clk  "WEIGHT 9";
TERMPLACE  clk  "CLOCK LEFT clk_l RIGHT clk_r";
TERMPLACE  x "TOP";
TERMPLACE  y "TOP";
TERMPLACE  in "TOP";
TERMPLACE  reset "TOP";
TERMPLACE  yout "BOT";
TERMPLACE  xout "BOT";
TERMPLACE  z "BOT"
TERMPLACE  rout "BOT";
```

The above control lines mean that the highest priority is assigned to the *clk* signal in the routing process and an additional channel is allocated for the *clk* signal. The input port terminals *x*, *y*, *in*, and *reset* are to be placed on the top side of the layout, and the output port terminals *yout*, *xout*, *z*, and *rout* are to be located at the bottom of the layout.

7.4.3 Layout by Placement and Routing

Complete standard cell layouts are acquired by using the cell script *sc_lpar*, which requires all the files generated so far. The script includes *sc_build*, *AutoCells* placement and routing, and *Lc* programs. It creates a set of standard cell layout or bounding boxes. The *AutoCells* placement and routing program is used to create an L program for the final layout. *Lc* is used to compile that program into a simple L file (in ASCII or binary format). The shell command of *sc_lpar* that is used to generate the multiplier is shown below.

```
sc_lpar -t scmos -x -g gdt.ctl -ROW 9 pdm.R

where
        -t scmos          : Technology file is the scmos
        -x                : Produces binary format output file
        -g gdt.ctl        : The standard cell control file is gdt.ctl
        -ROW 9            : The number of rows in the top level layout
                            is 9
        -pdm.R            : The routing file name is pdm.R
```

The above cell command produces the default output file *pdm_route.X*, which can be used in the GDT graphic editor *Led* to simulate and convert into the CIF file format.

7.4.4 Converting the Layout into a CIF File in Led

The shell command invoking *Led* with the final layout for the multiplier is shown below:

```
Led -t scmos pdm_route.X &
```

The Led screen consists of two windows. The layout appears in window 2 and shows just the first level of the layout, that is, the terminals and outlines of the rows. To see the entire levels, press the "p" key and modify the *plot depth* to 3, then hit the return key. After that, the *tab* key should be pressed to redraw the layout. Figure 18 shows the Led screen snapshot of the layout.
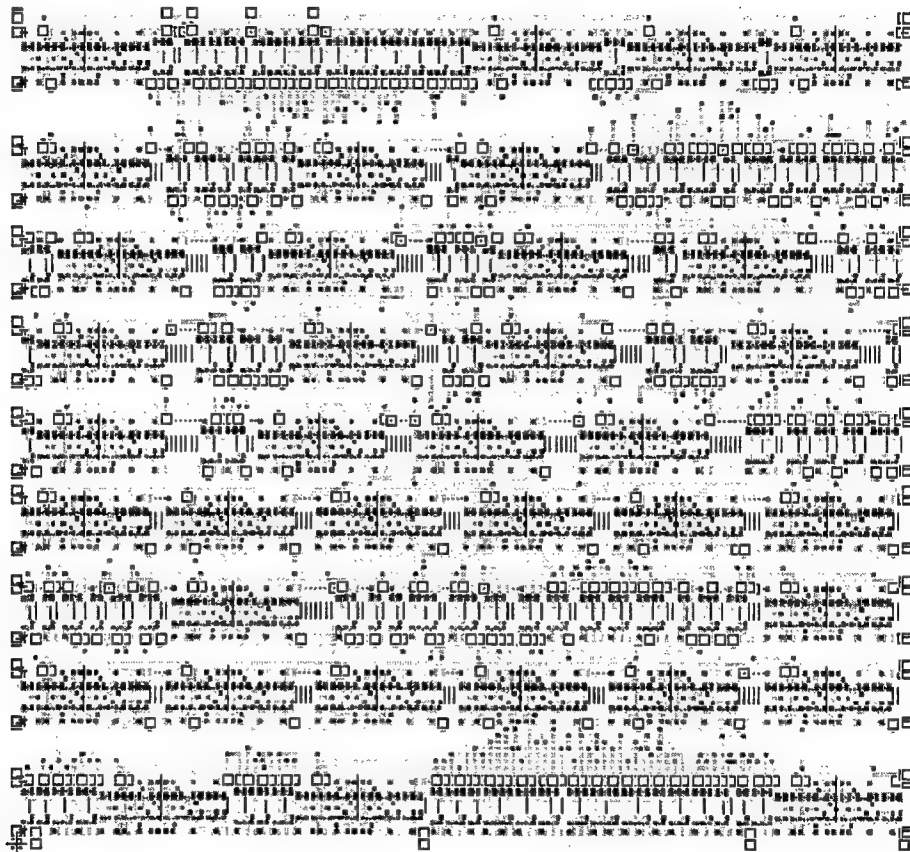
Figure 18: GDT Layout of the Multiplier

There are two kinds of menus, foreground and background, in the Led graphic editor. The background menu is used to exit Led, change screen color and cursor type, and so on. For the other facilities, the foreground menu is used. To invoke the

67

background menu, place the cursor on the gray part that does not belong to the two windows, and press the right mouse button. A detailed description of Led is beyond the scope of this paper. For further information, please refer to the Led Graphics Editor User Guide [10].

To convert the layout into a CIF file, invoke the foreground menu by pressing the right mouse button on the window area and selecting the *Utility* item. Then choose CIF submenu (or press 6), and change the mode to WRITE by clicking the READ icon, and give the file name. To exit Led, go to the background menu and select EXIT, or press the Ctrl+D keys.

8   L2MCIF — XY Mask Translation Compiler

This section describes the functionality of the *L2MCIF* compiler. This compiler is the critical link between the physical layout and IC fabrication. The physical standard cell layout is created by the GDT layout compiler tool suite using SCMOS technology. The layout should be prepared for mask fabrication at the MOSIS IC fabrication facility. MOSIS supports the SCMOS process technology and also provides compatible I/O pads and pre-laid out pad frames to perform the final chip assembly. These I/O pads and pad frames are readable and can be edited using the MAGIC tool. The HOL2GDT methodology supports and integrates MAGIC as the final chip assembly tool that assembles the standard cell physical layout from GDT to the I/O pad frames supplied by MOSIS. Global routing among the layout, pad cells, and power rails is also performed in MAGIC. To accomplish this, another complier is needed to transfer the GDT layout to the MAGIC layout.

The L2MCIF compiles the XY layout layer mask from GDT to the mask that is readable and reproducable in MAGIC. The layout mask is extracted from GDT in the Caltech Intermediate Form (CIF). CIF is an industry-wide standard layout description language used to transfer mask-level layouts between design tools. An alternative layout mask format is GDS II. The basic difference between GDS II and CIF is that GDS II is a binary-compiled format, and CIF is an ASCII format. The ASCII format provides easy readability and editability to mask compilers such as the L2MCIF.

A CIF XY layout mask consists of specific code-words for different layers (metal1, metal2, N-diffusion, etc). Each section in the mask specifies the code word for a particular mask layer, followed by X-axis and Y-axis co-ordinates that depict the area over which the layer is laid out. Below is an example of an XY mask for the

68

polysilicon layer (L POLY). Each line specifies a rectangular box (B), followed by length (L) and width (W) dimensions of the box and the XY coordinates for the center location of the box. For more details on CIF language syntax and semantics, please refer to the GDT Standard Cell Library [4].

```
Definition Start 1 100 1;
  L POLY;
  B L 3 W 3 Center 60, 6;
  B L 3 W 19 Center 60, -1;
  B L 3 W 3 Center 24, 6;
  B L 3 W 19 Center 24, -1;
  B L 3 W 3 Center 12, 6;
  B L 3 W 19 Center 12, -1;
Definition Finish;
```

The need for a mask compiler like the *L2MCIF* becomes necessary, because the CIF mask extracted from GDT layout tools has different mask-layer code words and syntax to specify XY coordinates than those supported by the MAGIC tool. The primary function of the L2MCIF is to compile the CIF mask of the GDT into a CIF mask of MAGIC so that we can utilize the ready-made pad design available in MAGIC. This can be done by mapping the layout layer code-words and appropriate language syntax between the two CIFs.

The L2MCIF compiler performs a dual-pass operation on the CIF mask generated by GDT tools. On the first pass, the L2MCIF does mask-layer codeword mapping between GDT and MAGIC. The second pass performs three functions: (1) maps the different contact types in GDT CIF mask to the appropriate combination of MAGIC CIF mask layers, (2) sets the minimum feature size for the contact dimensions and spacing between layer, (c) rewrites the mask data using the CIF syntax acceptable to MAGIC.

The design rules for the contact dimensions and minimum spacing between layers are different from each other and this causes design rule errors when mask layout is ported from GDT to MAGIC. The L2MCIF compiler attempts to minimize and correct some of these design rule errors during the compilation process. Since the compiler is not intended to be a design rule checker, design rule errors occur in complicated layouts, which can be corrected by using the MAGIC design rule checker after the mask layout is ported from GDT and recreated in MAGIC.

Table 1: CIF mask layer mapping between GDT and MAGIC

GDT CIF levels	SCMOS levels	Magic CIF Level
NSUB	N well substrate layer	CWN
PSUB	P well substrate layer	CWP
NPLUS	N plus implant mask	CSN
PPLUS	P plus implant mask	CSP
NDIFF (type I)	ACTIVE mask layer	CAA + CSN for contacts and transistors
NDIFF (type II)		CAA for other cases
NDIFF1 (type I)	ACTIVE mask layer	CAA + CSN for contacts and transistors
NDIFF1 (type II)		CAA for other cases
PDIFF	ACTIVE mask layer	CAA
PDIFF1	ACTIVE mask layer	CAA
POLY	POLYSILICON layer	CPG
MET1	METAL1 layer	CMF
MET2	METAL2 layer	CMS
CUTMD	METAL1 to ACTIVE layer contact	CCA
CUTMP	METAL1 TO POLYSILICON layer contact	CCP
CUTMM	VIA METAL1 TO METAL2 layer	CVA
GATE	NO EQUIVALENT	NO EQUIVALENT
LEV	NO EQUIVALENT	NO EQUIVALENT
MARKER	NO EQUIVALENT	NO EQUIVALENT
ICONLEV	NO EQUIVALENT	NO EQUIVALENT
NO EQUIVALENT	OVERGLASS	COG

An illustration of mask-layer mapping is shown in Table 1. Both GDT and MAGIC use the SCMOS (Scalable CMOS) technology. The only difference is that different layout mask-layers are denoted by unique keywords in each tool. The left column shows all the layer keywords supported and generated by the GDT tools. The center column displays the equivalent keywords used by the SCMOS technology. Finally, the right column shows the equivalent keywords supported by the MAGIC tool.

A unique case is the $NDIFF$ and $NDIFF1$ GDT mask layer, which is mapped as an $ACTIVE$ diffusion layer in SCMOS technology. These layers have two distinct mappings based on the place to which the GDT cell is to be mapped. If the cell being mapped is a contact or transistor, then the $NDIFF$ layers are mapped as a combination of two mask-layers, CAA and CSN, in MAGIC. The CAA is the active mask-layer and CSN is the $N\ PLUS\ SELECT$ N-plus implant mask-layer for the SCMOS technology. The N-plus implant mask-layer, along with the active mask-layer, creates the required N-diffusion layer. If the cell is mapped to anything other than a contact or transistor, then the $NDIFF$ layers are mapped as a single mask layer CAA in MAGIC. The GDT tools have two types of P-diffusion (PDIFF, PDIFF1) and N-diffusion layers (NDIFF, NDIFF1) to provide more controllability during CIF mask creation. The L2MCIF compiler maps the $PDIFF$ and $PDIFF1$ layers to the single MAGIC mask layer, CAA, and both $NDIFF$ and $NDIFF1$ layers are mapped to the same combination of CAA and CSN mask layers.

Table 2: Length x Width Summary for Contacts

GDT contact name	GDT Mask Layers	MAGIC contact type	MAGIC contact size
M1M2 - Metal1 Metal2 Contact	MET1, MET2, CUTMM	m2c	4x4
MPOLY - Metal1 Poly Contact	MET1, POLY, CUTMP	pc	4x4
MPDIFF - Metal1 P-Diffusion Cont.	MET1, NSUB, PPLUS, PDIFF, CUTMD	pdc	4x4
MNDIFF - Metal1 N-Diffusion Cont.	MET1, NDIFF, CUTMD	ndc	4x4
MNSUB - Metal1 N-Substrate Cont.	MET1, NSUB, NDIFF1, CUTMD	nsc	4x4
MPSUB - Metal1 P-Substrate Cont.	MET1, PPLUS, PDIFF1, CUTMD	psc	4x4

Certain mask layers ($GATE$, LEV, $MARKER$, $ICONLEV$) supported by GDT have no equivalent layers in MAGIC. Any specification for these layers in the GDT CIF mask is scrubbed out by the L2MCIF compiler. On the other hand, the MAGIC mask layer COG has no equivalent layer in GDT.

During the second pass of the L2MCIF compiler, the GDT CIF specifications for different contact types are mapped to a combination of multiple MAGIC CIF mask layers. The mapping for the contacts is illustrated in Table 2. The GDT CIF specifications for the contacts have different minimum dimensions that might cause design rule errors when they are mapped to MAGIC CIF mask. These design rule errors can be avoided by the L2MCIF compiler by mapping the GDT CIF specification for the contacts to the MAGIC CIF contact specification, as shown in Table 2.

The syntax of the L2MCIF compiler is as shown below.

```
l2mcif <input file: gdt cif> <output file: magic cif>
```

The output of L2MCIF is a MAGIC readable CIF file. This CIF file can be read by invoking an empty MAGIC window with a new name and specifying the CIF read command within the MAGIC shell as shown below.

```
cif read <magic cif file>
```

The *cif read* command recreates the layout in the MAGIC layout editor window. To save all subcomponents in the hierarchy, the MAGIC command **write all** should be executed. For more details about MAGIC commands, please refer to the MAGIC manual [8]. The MAGIC layout retrieved from CIF file with all errors corrected is shown in Figure 19.

Once the layout in MAGIC is edited to resolve design rule errors, the final chip assembly of the layout and I/O pad frame is performed. First, the global routing

71
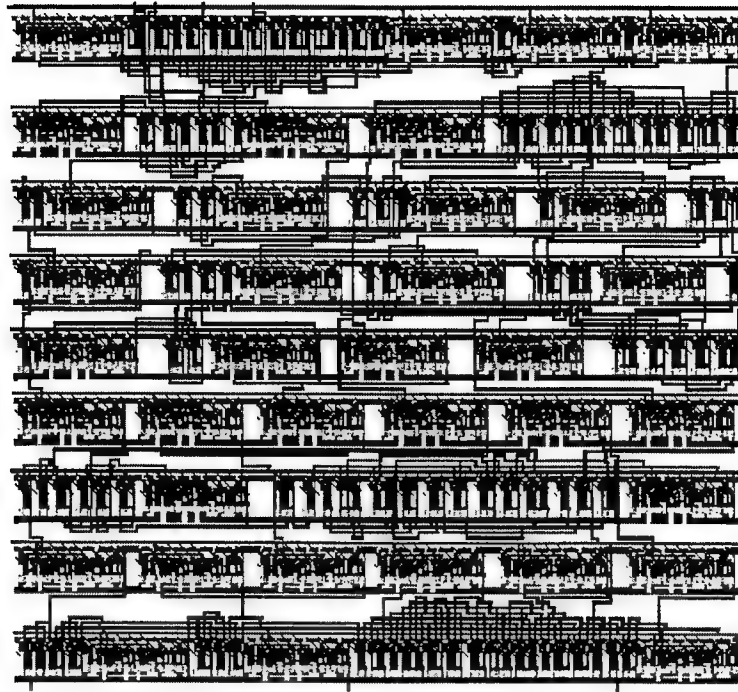
Figure 19: MAGIC Layout of the Multiplier

of the primary input and output ports of the layout to the respective input and output pad cells is performed. Then, global power rail routing is performed. The final assembled layout for the multiplier is shown in Figure 20.

9 Functional Testing via Multi-Level Simulation

The HOL2GDT and L2MCIF compilers facilitate functional simulation and testing of the design using multiple simulators. In current methodology three simulators are involved:

- Mentor Graphics Lsim mixed-level simulator

- MAGIC supported IRSIM simulator

- SPICE3 simulator
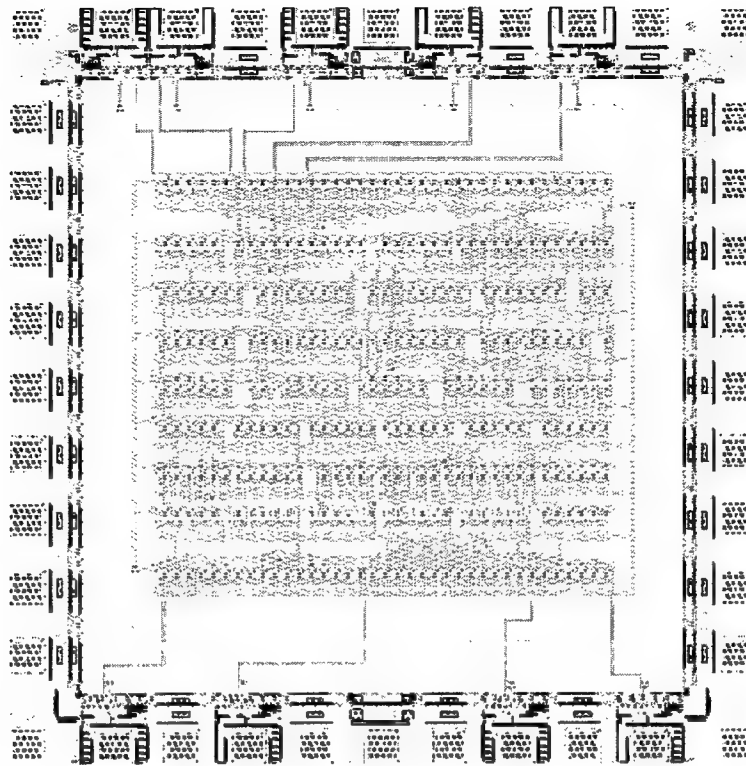
Figure 20: MAGIC Layout after Pad Frame Assembly

9.1 Mentor Graphics Lsim Simulator

The *Lsim* simulator is a mixed-level simulator that supports functional, gate, and switch level simulations. The Lsim simulator is an integral component of the Mentor Graphics GDT design system. It requires two input files, the extracted layout netlist file and the vector stimulus file. The *Led* layout editor provides a menu-driven utility that extracts the design netlist from the standard cell layout and translates it into a format supported by Lsim. The utility prompts for an output filename to write out the netlist. The netlist file has an *.N* suffix. The vector stimulus file has to be built and supplied by the user. The vector file is also known as the initialization file, as it initializes the inputs of the circuit before the simulator starts. The vector file has a specific format that consists of three sections.

The first section starts with the Lsim command **order**. This section lists the order of the primary input and output ports of the design. The simulator graphic interface

73

displays the result of the simulation as waveforms of the I/O ports and lists them in the order provided by the order command. An example of an order command for the serial multiplier design is shown below:

```
order clk reset y x in yout xout z rout
```

The second section of the vector file is an initialization section. The primary input signals like *clock* and *reset* are initialized at the start of the simulator. An example of signal intialization is shown below:

```
pwl clk 0,l  5,t  10,t  repeat 0
s 20
h reset
s 20
```

The above example shows how a clock signal called *clk* is initialized for 10 ns clock period. Here, **pwl** and **repeat** are Lsim simulator commands. The pwl command is an acronym for *piecewise linear*. The first line of the initialization section defines a piecewise linear clock signal named *clk* that is low (logic 0) at 0 ns, toggled to high at time 5 ns, and toggled to low at 10 ns. This waveform is repeated from the time 0 ns. The next line in the initialization section is *s 20*, where *s* is an acronym for the *step* command. This command line lets the simulator advance 20 ns. The next line sets the reset signal to high (logic 1), which would reset the circuit under simulation. Finally, the last line performs a step command to advance 20 ns. The user can also initialize any tied-low or tied-high signals in this section.

The third section is a vectors section, in which a vector stimulus (low, high or don't-care) is applied to input signals, followed by a step command. Below is an example of a vector section:

```
l reset y
h x in
s 10

x reset y x in
s 10
```

The first vector sets the *reset* and *y* signals to low and the *x* and *in* signals to high. The input signals can also be set to don't-care value (logic X).

Note that if the vector file is not available or not supplied to the simulator, then the Lsim is invoked in the interactive mode. In this mode the signals are initialized interactively. The vector file is required only if the simulator is to be run in batch or regression-testing mode. The vector file usually has an *.i* suffix. Once the vector stimulus file is completed, the Lsim simulator is invoked with the following command line syntax:

```
Lsim -i pdm_vect.i pdm.N
```

The *-i* parameter indicates that Lsim will get the simulator startup commands from the *pdm_vect.i* file and *pdm.N* is the extracted netlist file from the Led layout editor.

The result of the Lsim simulation is shown in Figure 21.
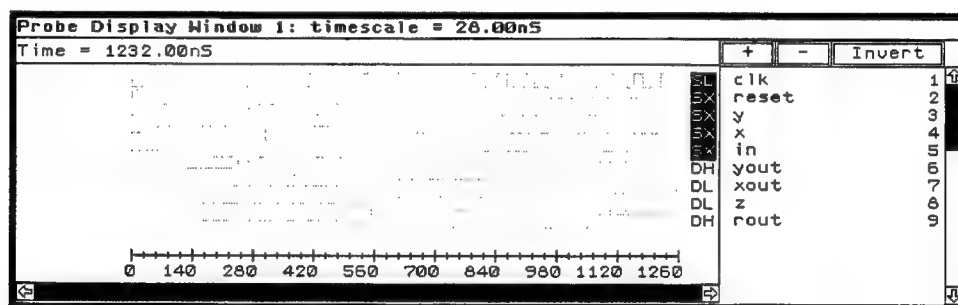
Figure 21: The Result of Lsim Simulation

9.2 IRSIM Simulation

IRSIM is an event-driven, logic-level simulator for MOS (both N and P) transistor circuits. This simulator is used to simulate the functionality of the circuit after the layout is converted from the GDT format into MAGIC format. There are two simulation models in the IRSIM, the **switch** model and the **linear** model, which are determined by the method of modeling the transistors in the circuit. In the switch model, each transistor is modeled as a voltage-controlled switch. It is useful for initializing or determining the functionality of the network. In the linear model,

75

each transistor is modeled as a resistor in a series with a voltage-controlled switch a capacitance. Node values and transition times can be computed from the RC network.

9.2.1   Syntax of the IRSIM

The synopsis of the IRSIM is as below:

```
irsim [-s] prm_file  sim_file  [+hist_file]  [-cmd_file]
```

If the -s switch is specified, two or more transistors of the same type are to be connected in series, and no other connections to their common source/drain will be stacked into a compound transistor with multiple gates.

The **prm_file** is the electrical parameters file that configures the devices to be simulated. It defines the capacitance of various layers, transistor resistances, and threshold voltages. The parameter file usually has a suffix of *.prm*. The **sim_file** is a file to be simulated that contains three types of information: environmental information (scaling, timestamps, etc.), the extracted circuit corresponding to the mask geometry of cells in the circuit, and the connections between this mask geometry and the subcells of the circuit. The sim_file is a hierarchical netlist of the circuit and is obtained by using MAGIC's extractor **EXT**. The MAGIC command **ext** will generate the netlist, and by default the output file name has *.ext* suffix. The extracted circuit is converted to a flat *sim* file by the *EXT2SIM* program. This flattened extract netlist file can be used by many simulation tools such as *Crystal*, *sim2spice*, and *IRSIM*. The basic syntax of the EXT2SIM is shown below:

```
ext2sim [-o simfile] root-file
```

This program runs just with the root-file name that has an *.ext* suffix. Since it is the root of the tree to be extracted, all files it references are recursively flattened. The flat representation of the circuit is written in the file *root.sim*. In addition, two additional files, *root.al* and *root.nodes*, are generated by default for the node aliases and the locations of all node names in CIF format, respectively.

For the IRSIM syntax again, the file names prefaced with a hyphen are assumed to be the *command files* that contain command lines to be processed in the normal

76

fashion. These files are processed line by line, and if the last command of the file is not an *exit*, IRSIM will accept further commands from the user.

The actual command line used to simulate the multiplier is shown below:

```
irsim  s.prm  pdm.sim -pdmtest
```

The *s.prm* is the parameter file, *pdm.sim* is the output from the ext2sim program, and *pdmtest* is the command file. The s.prm and pdmtest files are listed in the appendix.

The IRSIM commands used to test the multiplier are **h**, **l**, **x**, **s**, and **ana**. The commands **h** and **l** make nodes logic high (1) and logic low (0), respectively. The command **x** removes nodes from the display list, and the command **s n** simulates for n ns, in which n is a step size. Finally, the command **ana** displays all nodes in the analyzer window.

9.2.2 Analysis of the IRSIM Simulation

The simulation result of the IRSIM is shown in Figure 22.
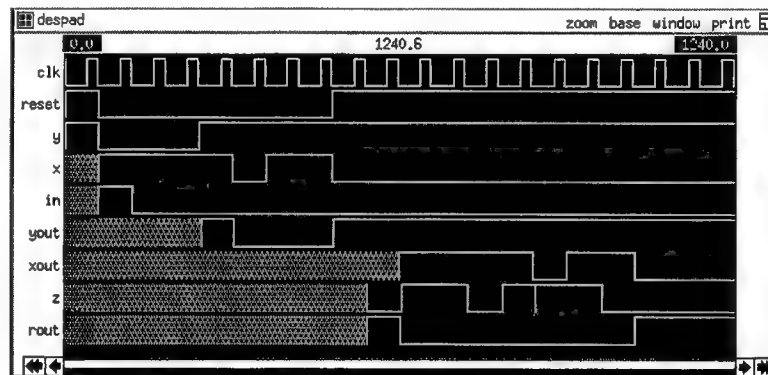
Figure 22: The Result of IRSIM Simulation

The *clk*, *reset*, *y*, *x*, and *in* are the input signals, and *yout*, *xout*, *z*, and *rout* are the output signals. The x signal is the multiplicand, and y is the multicator. The multiplier in the simulation is five-stage, thus the multiplicator is a five-digit binary number. In this simulation, the least significant bit enters first, hence the value of

77

the multiplicand x becomes 1101111_2 and the y value becomes 10001_2. The result of the multiplication is represented by the signal z, which has a value of 111011_2. The z value is extracted only during the period when *rout* value is at low state. The first 0 value of z is ignored. In the decimal system, the multiplicand x is 111 and multiplicator y is 17. The correct output value will be 111 * 17 = 1887. The z value 59 represents the closest value for the division of the x * y by 2^5.

9.3   SPICE Simulation

SPICE is a general-purpose circuit simulation program for nonlinear DC, nonlinear transient, and linear AC analyses. Circuits may contain resistors, capacitors, inductors, mutual inductors, independent voltage and current sources, four types of dependent sources, lossless and lossy transmission lines, switches, uniformly distributed RC lines, and the five most common semiconductor devices: diodes, BJTs, JFETs, MESFETs, and MOSFETs. In HOL2GDT methodology, the SPICE version 3f4 was used.

9.3.1   History of Using the SPICE Program

The first multiplier chip we built did not work properly. That is, the *rout* signal's 0 value duration was one clock cycle shorter than expected as shown in Figure 22. After investigating the multiplier layout, we suspected that there might be a **clock skew** between the third and fourth FFs, and if the skew is greater than 6 ns, that could explain the malfunction of the multiplier chip. Hence we decided to verify the clock skew problem by using the SPICE3 simulator. However, the SPICE3 simulator was not powerful enough to simulate the entire multiplier circuit. For example, a transient analysis for a single FF would take 10 to 15 minutes. The multipler contains 36 FFs as well as many other components. It would take unreasonably long time to simulate the multiplier. Since the main purpose of the simulation was to calculate the time gap between the clock inputs to FF3 and FF4, we cut down the layout (Figure 23). That is, we removed other lines and components except for the clock net and FFs, and even the FFs were shrunken as inverters.

Another problem was waiting for us. Because SPICE3 considered all nodes that were connected with metal or poly lines to be the same node, we could not get any delay difference at all among all clock input nodes of the FFs. We suspected that there would be a clock signal delay between the FF3 and the FF4 because the path length from root clock signal to the FF4 was much longer than the length from the

78
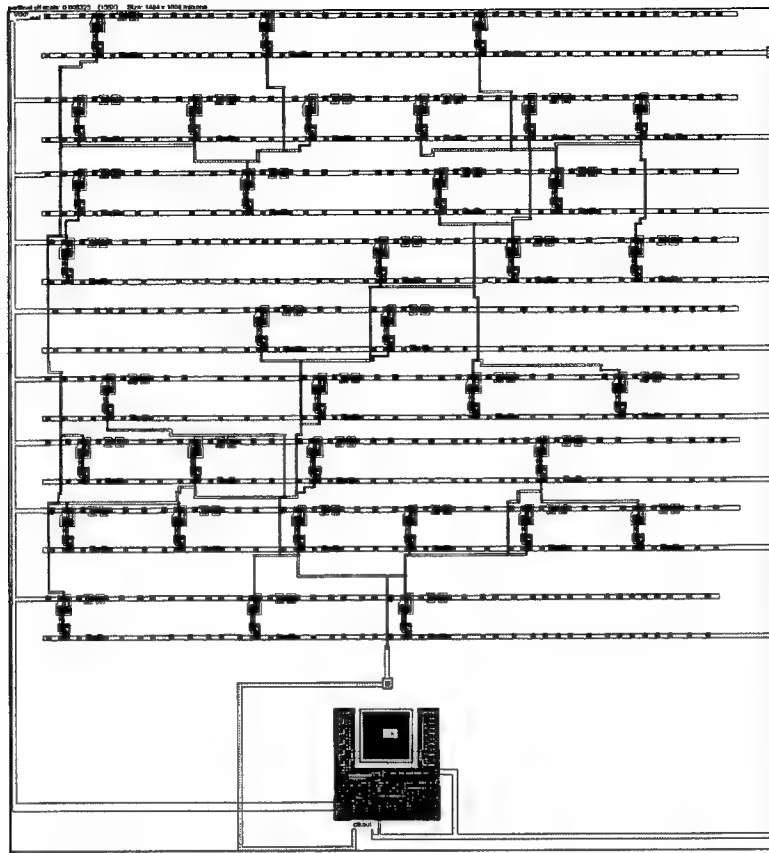
Figure 23: Simplified Multiplier Circuit for SPICE Simulation

root to the FF3. However, if simulator3 considered those two input nodes to be the same one, there was no way to get the signal delay. We next modeled the clock net by manually replacing all the poly feed lines with resistors so that the simulator could recognize the FF3 and FF4 clock signal input nodes as being different. In modeling the circuit we made use of all the information from the ext and ext2spice program. That is, the resistance value for the poly feed line was extracted from the spice file that the ext2spice program produced. However, the capacitance values of the poly feed lines were ignored because their values were ignorably small. For the MOS model, the process parameters for actual manufacturing were used. (The MOS transistor model is attached in the appendix). The modeled circuit is shown in Figure 24.
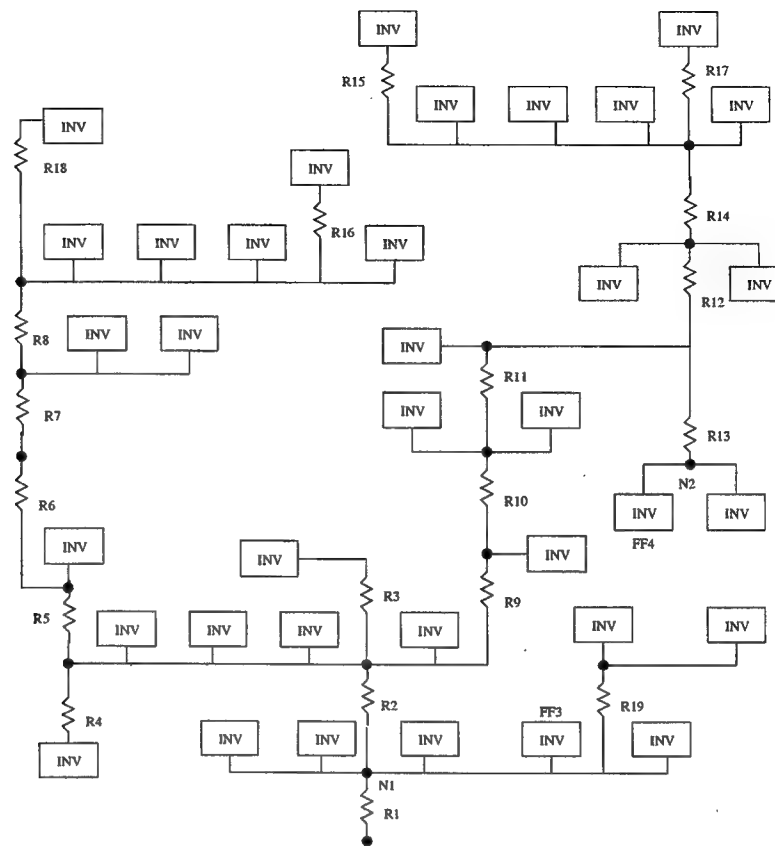
Figure 24: Modeled Clock Net

9.3.2 The Simulation Procedures and the Results

To get the SPICE3 input file, two programs should be executed. First, the MAGIC's extraction program *EXT* is used to get the hierarchical netlist, which is then used as the input to the *ext2spice* program to get the spice input file. The basic syntax of the ext2spice program is shown below.

```
ext2spice [-o outfile] [-Cxxx] [-R] root_file
```

The root_file is the output of the MAGIC **ext** command that has the .*ext* suffix. Without the [-o outfile] option, the spice file will generate the file name, *root_file*

80

.*ext*. The [-Cxxx] option will supress all the node capacitance values below xxx and the [-R] option will supress all the internode resistances. The [-Cxxx] option does not make any noticeable difference to the result, because the capacitance values are insignificantly small. However, [-R] option makes severe difference to the result. The actual command used to get the spice input file is shown below.

```
ext2spice -R modeled.ext
```

This command produces the *modeled.spice* file, which is to be modified to include MOS model parameters, a DC power source, and the clock input signal. The final spice input file is listed in the appendix.

Finally, the SPICE3 program is invoked by command below:

```
spice3  modeled.spice
```

If there is no error in the model.spice file, the SPICE3 program gives a prompt. The first step is to run the file by issuing a **run** command. The second step is to do a transient analysis by issuing a **tran** command. The syntax of the tran command is like below:

```
tran tstep tstop [tstart [tmax]]
```

Tstep is the printing or plotting increment for the output. If the *tstep* value is too small, then the simulation will take a long time to get the result. On the other hand, if the *tstep* value is too large, some details of the simulation will be lost. *Tstop* is the final time, and *tstart* is the initial time. If *tstart* is omitted, it is assumed to be zero. The transient analysis always begins at time zero. *Tmax* is the maximum step size that SPICE3 uses. For the default, the program chooses either *tstep* or (*tstop - start*)/50.0, whichever is smaller. *Tmax* is useful when one wishes to guarantee a computing interval that is smaller than the printer increment, *tstep*.

The actual **tran** command we used is shown below:

```
tran 2ns 800 ns
```

81

The next command involved is **plot**. It is used to display the simulation result in graphic mode on the screen. The actual **plot** command being used is:

```
plot v(301) v(329)
```

The above command opens a window that shows the transient voltage values of the nodes numbered 301 and 329 in graphic mode. Node 301 corresponds to FF3, whose input is supplied through the resistor R1, and the node 329 corresponds to FF4 whose input is supplied through the resistors R1, R2, R9, R10, R11, and R13. The simulation result is shown in Figure 25.
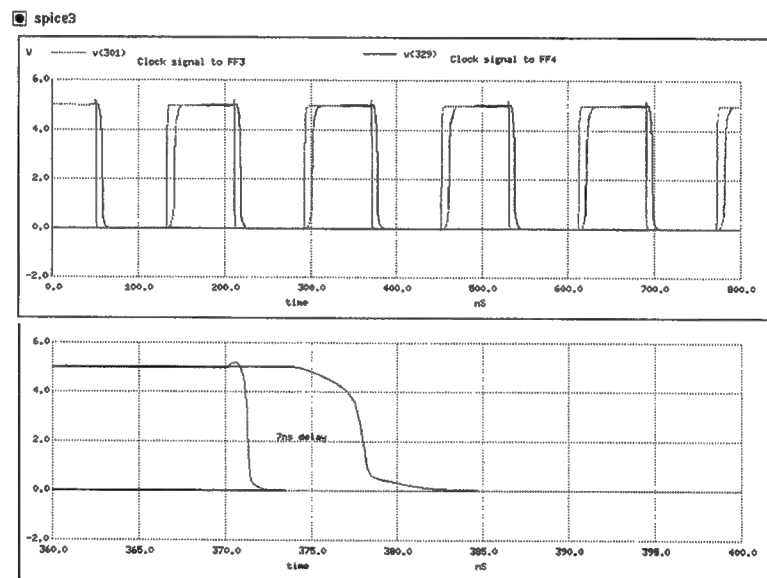
Figure 25: Delay between FF3 and FF4 Clock Signals

The simulator result reveals that there exists a time gap between the clock inputs of the FF3 and FF4 (the lower window shows that the time gap is around 7ns). With the SPICE3 simulation the clock skewing problem can be determined, and this skewing is solved by adding additional metal clock net in each row.

10 Testing of the Actual Chip

Figure 26 shows the actual fabricated chip.
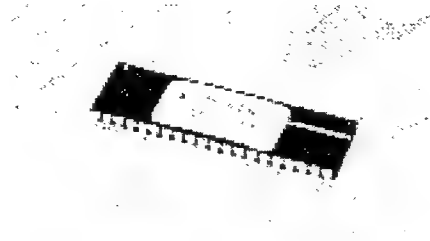
Figure 26: The Multiplier Chip

For the final test to check whether the chip functions correctly, a Hewlett Packard Logic Analysis System model 16500A (shown in Figure 27) is used. The screen of the analyzer show the testing result.
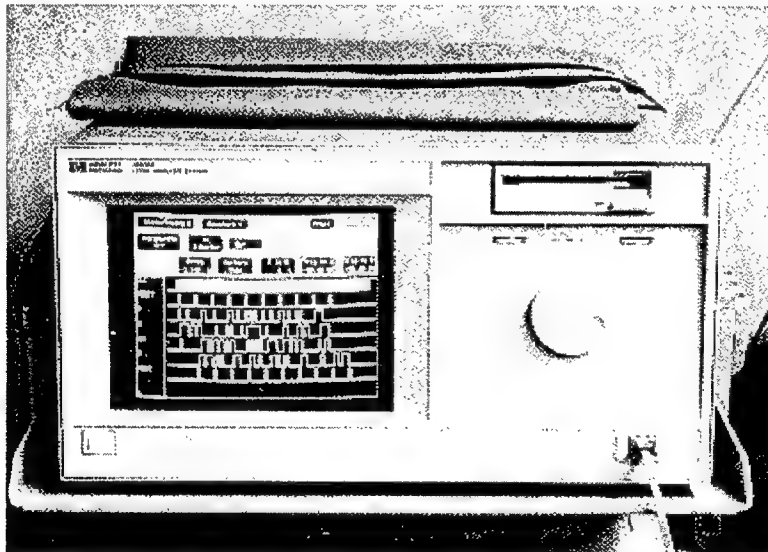
Figure 27: Hewlett Packard Logic Analysis System 16500A

11 Conclusions

In this report we explained the HOL2GDT methodology by describing how to define formal implementation descriptions of the hardware design, how to translate implementation descriptions into L language schematic generator models, and how to get physical IC layouts from schematic models. A complete example of an n-bit serial multiplier design was used to illustrate the HOL2GDT design methodology. Since the implementation description of the multiplier is formally verified and the layout is generated from the verified description, we believe in the correctness of the multiplier chip. The multiplier example is a detailed illustration of the HOL2GDT methodology. Currently we are working on the implementation of the Data Encrytion Standard (DES) algorithm on Xilinx's FPGA using the HOL2GDT methodology.

References

[1] Tom Melham, Higher Order Logic and Hardware Verification, Cambridge Tract in Theoretical Computer Science 31.

[2] Juin-Yeu Lu, Shiu-Kai Chin, Linking HOL to a VLSI CAD System, CASE Center Report No. 9308, May 1993.

[3] Shiu-Kai Chin and Juin-Yen Lu, The Mechanical Verification and Synthesis of Parameterized Serial/Parallel Multiplier, CASE Center Tech. Report No. 9140, Syracuse University, March 1991.

[4] Mentor Graphics, Inc., GDT Standard Cell Library User Guide version 5. May, 1991.

[5] R. Kumar, K. Schneider, T. Kropf, Structuring and Automating Hardware Proofs in a Higher-Order Theorem-Proving Environment, *Formal Methods in System Design*, 2, pp. 165–223, 1993.

[6] C. Mead and L. Conway, Introduction to VLSI systems, Addison-Wesley, 1980.

[7] Mentor Graphics, Inc., L Compiler User Guide version 5, May 1991

[8] Gordon Hamachi, Robert Mayo, Magic Manual version 6.3, Western Research Laboratory, September 1990.

[9] University of Berkeley, SPICE3 Users Manual, version 3f4, August 1995.

[10] Mentor Graphics, Inc., Led Graphics Editor Users Guide version 5, May 1991

Appendix

A   Basic Gates Definition File

```
(* Behavioral definitions of the primitive gates for HOL2L compiler  *)
(* Updated 2/20/95; types ":sig" and ":toggle" are removed.          *)
(* Updated 4/21/95; definitions of master-slave FFs                  *)
(* Updated 11/7/96; transferred to HOL90.7                           *)

load_theory  "string";
new_theory "gates";


val port_width = define_type
    {fixities = [Prefix, Prefix],
     name = "port_width",
     type_spec = 'port_width = pin | bus of num'};

val port = define_type
    {fixities = [Prefix, Prefix, Prefix, Prefix],
     name = "port",
     type_spec = 'port = in_port of string => port_width
                       | out_port of string => port_width
                       | inout_port of string => port_width
                       | cell_size of string'};

val inv = new_definition
    ("inv",
    (--' inv(a:num->bool, b) =
        !t. b t = ~(a t)'--));

val or = new_definition
    ("or",
    (--'or(x:num->bool, y, out) =
        !t. out t = x t \/ y t'--));

val or3 = new_definition
    ("or3",
    (--'or3(x:num->bool,y,z,out) =
        !t:num. out t = x t \/ y t \/ z t'--));

val or4 = new_definition
```

```
    ("or4",
    (--'or4(w, x, y, z, out) =
        !t:num. out t = w t \/ x t \/ y t \/ z t'--));

val or5 = new_definition
    ("or5",
    (--'or5(x:num->bool,y,z,w,q,out) =
        !t. out t = x t \/ y t \/ z t \/w t \/q t'--));

val and2 = new_definition
    ("and2",
    (--'and2(x:num->bool, y, out) =
        !t. out t = x t /\ y t'--));

val and3 = new_definition
    ("and3",
    (--'and3 (x:num->bool,y,z,out) =
        ! t.out t = x t /\ y t /\ z t'--));

val and4 = new_definition
    ("and4",
    (--'and4 (x:num->bool,y,z,w,out) =
        ! t.out t = x t /\ y t /\ z t /\ w t'--));

val and5 = new_definition
    ("and5",
    (--'and5 (x:num->bool,y,z,w,q,out) =
        ! t.out t = x t /\ y t /\ z t /\ w t /\ q t'--));


val xor = new_definition
    ("xor",
    (--'xor(x:num->bool, y, out) =
        !t. out t = (~x t /\ y t) \/ (x t /\ ~y t)'--));

val nand = new_definition
    ("nand",
    (--'nand(x:num->bool, y, out) =
        !t. out t = ~(x t /\ y t)'--));

val nand3
 = new_definition
    ("nand3",
    (--'nand3(in0, in1, in2, out) =
```

```
                !t:num. out t = ~(in0 t /\ in1 t /\ in2 t)'--));

val nand4 = new_definition
     ("nand4",
     (--'nand4(in0, in1, in2, in3, out) =
          !t:num. out t =
             ~(in0 t /\ in1 t /\ in2 t /\ in3 t)'--));

val nand5 = new_definition
     ("nand5",
     (--'nand5(x:num->bool,y,z,w,q, out) =
          !t. out t =
             ~(x t /\ y t /\ z t /\w t /\q t)'--));

val nor = new_definition
     ("nor",
     (--'nor(x:num->bool, y, out) =
          !t. out t =  ~(x t \/ y t)'--));

val nor3 = new_definition
     ("nor3",
     (--'nor3(in0, in1, in2, out)  =
          !t:num. out t =  ~(in0 t \/ in1 t \/ in2 t)'--));

val nor4 = new_definition ("nor4",
     (--'nor4(in0, in1, in2, in3, out)  =
          !t:num. out t =
             ~(in0 t \/ in1 t \/ in2 t \/ in3 t)'--));

val nor5 = new_definition
          ("nor5",
          (--'nor5(x:num->bool,y,z,w,q,out) =
               !t. out t =
                  ~(x t \/ y t \/ z t \/ w t \/ q t)'--));

val d_reg = new_definition
     ("d_reg",
     (--'d_reg(din:num->bool, clk, reset, q) =
          !t. (reset t ==> ~q t) /\
              (~reset(SUC t) /\ clk(SUC t) ==>
                            (q (SUC t) = q t)) /\
              (~reset(SUC t) /\ ~clk t /\ ~clk(SUC t) ==>
                            (q (SUC t) = q t)) /\
              (~reset t /\ ~reset(SUC t) /\ clk t /\ ~clk(SUC t)
```

```
                        ==> (q(SUC t) = din t)) /\
                (~din t /\ clk t /\ ~clk(SUC t) ==> ~q(SUC t))'--));

    val d_reg_nr = new_definition
        ("d_reg_nr",
        (--'d_reg_nr(din:num->bool, clk, q) =
            !t. (clk(SUC t) ==> (q (SUC t) = q t)) /\
                (~clk t /\ ~clk(SUC t) ==> (q (SUC t) = q t)) /\
                (clk t /\ ~clk(SUC t) ==> (q(SUC t) = din t))'--));

    val d_reg_wqb = new_definition
        ("d_reg_wqb",
        (--'d_reg_wqb(din:num->bool, clk, q, qb) =
            !t.(qb t = ~q t) /\
                (clk(SUC t) ==> (q (SUC t) = q t)) /\
                (~clk t /\ ~clk(SUC t) ==> (q (SUC t) = q t)) /\
                (clk t /\ ~clk(SUC t) ==>  (q (SUC t) = din t))'--));

    val d_reg_wrqb = new_definition
        ("d_reg_wrqb",
        (--'d_reg_wrqb(din:num->bool, clk, reset, q, qb) =
            !t. (qb t = ~q t) /\
              (reset t ==> ~q t) /\
              (~reset(SUC t) /\ clk(SUC t) ==> (q (SUC t) = q t)) /\
              (~reset(SUC t) /\ ~clk t /\ ~clk(SUC t) ==> (q (SUC t) = q t)) /\
              (~reset t /\ ~reset(SUC t) /\ clk t /\ ~clk(SUC t)
                    ==> (q(SUC t) = din t)) /\
              (~din t /\ clk t /\ ~clk(SUC t) ==> ~q(SUC t) /\ qb(SUC t))'--));

    val lsl_wr = new_definition
        ("lsl_wr",
        (--'lsl_wr(din:num->bool, phi, phib, reset, q, qb) =
            !t.(qb t = ~q t) /\
                (reset t ==> ~q t) /\
                (~reset t /\ phi t /\ ~phib t ==> (q t = din t)) /\      (* Flush *)
                (~reset(SUC t) /\ ~phi t /\ ~phi(SUC t) /\ phib t /\ phib(SUC t)
                    ==> (q(SUC t) = q t)) /\                             (* Hold  *)
                (~reset t /\ ~reset(SUC t) /\ phi t /\ ~phi(SUC t) /\
                 ~phib t /\ phib(SUC t) /\ (din(SUC t) = din t)
                    ==> (q(SUC t) = din t)) /\                           (* Sample *)
                (~din t /\ ~din(SUC t) /\ ~reset(SUC t) /\ phi t /\
                 ~phi(SUC t) /\ ~phib t /\ phib(SUC t) ==> ~q(SUC t))'--));
                                                                        (* Zero *)
    val lsl_nr = new_definition
```

```
    ("lsl_nr",
    (--'lsl_nr(din:num->bool, phi, phib, q, qb) =
        !t.( qb t = ~q t) /\
        ( phi t /\ ~phib t ==> (q t = din t)) /\                    (* Flush *)
        (~phi t /\ ~phi(SUC t) /\
          phib t /\ phib(SUC t) ==> (q(SUC t) = q t)) /\            (* Hold *)
        ( phi t /\ ~phi(SUC t) /\
         ~phib t /\ phib(SUC t) /\ (din(SUC t) = din t)
           ==> (q(SUC t) = din t))'--));                           (* Sample *)

val pass = new_definition
    ("pass",
    (--'pass (d1:num->bool,d2, g, gb) =
        !t. g t /\ ~gb t ==> (d1 t = d2 t)'--));

val by_pass = new_definition
    ("by_pass",
    (--'by_pass (in1:num->bool, out) =
        !t:num. in1 t = out t'--));

val bus2wire = new_definition
    ("bus2wire",
    (--'bus2wire (abus:num->num->bool) (n:num) (abit:num->bool) =
        !t:num. abus n t = abit t'--));

val wire2bus = new_definition
    ("wire2bus",
    (--'wire2bus (abit:num->bool) (abus:num->num->bool) (n:num) =
        !t:num. abit t = abus n t'--));

val tbfi = new_definition
    ("tbfi",
    (--'tbfi (a:num->bool,out, g, gb) =
        !t. ((a t /\ g t) \/ (~a t /\ ~gb t))
            ==> (out t = ~a t)'--));

val buffer = new_definition
    ("buffer",
    (--'buffer(in1:num->bool,load,out) =
        ! t. out t = (load t => in1 (t) | out(t - 1))'--));

val buffer_s = new_definition
    ("buffer_s",
    (--'buffer_s(in1:num->bool,load,out) =
```

```
                  ! t. out t = (load t => in1(t) | out(t - 1))'--));

val vdd = new_definition
     ("vdd",
     (--'vdd(out) = !t:num. out t'--));

val gnd = new_definition
     ("gnd",
     (--'gnd (out) = !t:num. ~out t'--));

export_theory();
close_theory();
```

B   Macro Cell Definition File: pdm.macro

```
L:: TECH ANY
SCHEMATIC d_reg()
{

  IN in;
  IN clk;
  IN reset;
  OUT out;

  INST msff msff;

  WIRE msff.clk TO clk;
  WIRE msff.reset[0] TO reset;
  WIRE msff.reset[1] TO reset;
  WIRE msff.in TO in;
  WIRE msff.out TO out;
}

SCHEMATIC d_reg_nr()
{

  IN in;
  IN clk;
  OUT out;

  INST msff_nr msff_nr;
```

```
  WIRE msff_nr.clk TO clk;
  WIRE msff_nr.in TO in;
  WIRE msff_nr.out TO out;
}

SCHEMATIC d_reg_wqb()
{

  IN in;
  IN clk;
  OUT out;
  OUT out_b;

  INST msff_nr msff;

  WIRE msff.clk TO clk;
  WIRE msff.in TO in;
  WIRE msff.out TO out;
  WIRE msff.out_b TO out_b;
}

SCHEMATIC d_reg_wrqb()
{

  IN in;
  IN clk;
  IN reset;
  OUT out;
  OUT out_b;

  INST msff_wrqb msff_wrqb;

  WIRE msff_wrqb.clk TO clk;
  WIRE msff_wrqb.reset[0] TO reset;
  WIRE msff_wrqb.reset[1] TO reset;
  WIRE msff_wrqb.in TO in;
  WIRE msff_wrqb.out TO out;
  WIRE msff_wrqb.out_b TO out_b;
}

SCHEMATIC buffer (  )
{
# 3 terminals
# set load begin High to load the data from in
```

```
        IN in ;
        IN load ;
        OUT out ;


        INST latchdd latchdd ;
        INST inv inv ;

        WIRE in TO latchdd.in ;
        WIRE out TO latchdd.out ;
        WIRE inv.out TO latchdd.clk_b ;
        WIRE inv.in TO latchdd.clk ;
        WIRE load TO inv.in ;

}

SCHEMATIC buffer_s ()
{
# made by A. Chavan May 14 96
# 3 terminals
# set load begin High to load the data from in

        IN in ;
        IN load ;
        OUT out ;

        INST inv inv ;

        WIRE inv.in TO load ;
        SIG inv.out "11";

        INST lsl_nr lsl_nr ;

        WIRE lsl_nr.in TO in;
        WIRE lsl_nr.clk TO load;
        SIG lsl_nr.clk_b "11" ;
        WIRE lsl_nr.out TO out ;
}

SCHEMATIC vdd()
{

   OUT out;
```

93

```
    VDD v1;

    INST inv inv0;
    WIRE inv0.in TO v1;
    SIG inv0.out "11";

    INST inv inv1;
    SIG inv1.in "11";
    WIRE inv1.out TO out;
}
SCHEMATIC gnd()
{
    OUT out;
    VDD v1;

    INST inv inv;
    WIRE inv.in TO v1;
    WIRE inv.out TO out;
}

SCHEMATIC by_pass()
{
    IN in;
    OUT out;

    INST inv inv[0];
    WIRE inv[0].in TO in;
    SIG inv[0].out "11";

    INST inv inv[1];
    SIG inv[1].in "11";
    WIRE inv[1].out TO out;

}

SCHEMATIC pass()
{
    INOUT in;
    INOUT out;
    IN g;
    IN gb;

    TN tn0;
    TP tp0;
```

```
      CON c_in;
      CON c_out;
      CON c_g;
      CON c_gb;

      WIRE c_in TO in;
      WIRE c_in TO tn0.d;
      WIRE c_in TO tp0.d;
      WIRE c_out TO out;
      WIRE tn0.s TO c_out;
      WIRE tp0.s TO c_out;
      WIRE c_g TO g;
      WIRE tn0.gl TO c_g;
      WIRE c_gb TO gb;
      WIRE tp0.gl TO c_gb;

}

SCHEMATIC bus2wire()
{
   IN in;
   OUT out;

   INST inv inv[0];
   WIRE inv[0].in TO in;
   SIG inv[0].out "11";

   INST inv inv[1];
   SIG inv[1].in "11";
   WIRE inv[1].out TO out;

}

SCHEMATIC wire2bus()
{
   IN in;
   OUT out;

   INST inv inv[0];
   WIRE inv[0].in TO in;
   SIG inv[0].out "11";
```

```
   INST inv inv[1];
   SIG inv[1].in "11";
   WIRE inv[1].out TO out;

}
```

C   Printed Theory File for Pipelined Multiplier

```
|- cell_port_def =
   ['far',
     [in_port 'x' pin;in_port 'y' pin;in_port 'cin' pin;
      in_port 'reset' pin;out_port 'sum' pin;out_port 'cout' pin];
    'sa',
     [in_port 'x' pin;in_port 'y' pin;out_port 'sum' pin;
      in_port 'clk' pin;in_port 'reset' pin];
    'wff',
     [in_port 'in' pin;in_port 'w' pin;out_port 'out' pin;
      in_port 'clk' pin];
    'cell',
     [in_port 'x' pin;in_port 'y' pin;in_port 'in' pin;
      in_port 'reset' pin;in_port 'clk' pin;out_port 'xout' pin;
      out_port 'out' pin;out_port 'rout' pin];
    'bps',[in_port 'in' pin;out_port 'out' pin];
    'bcell',
     [in_port 'x' pin;in_port 'y' pin;in_port 'in' pin;
      in_port 'reset' pin;in_port 'clk' pin;out_port 'yout' pin;
      out_port 'xout' pin;out_port 'z' pin;out_port 'rout' pin];
    'icell',
     [in_port 'xi' pin;in_port 'yi' pin;in_port 'zi' pin;
      in_port 'ri' pin;in_port 'clk' pin;out_port 'yout' pin;
      out_port 'xout' pin;out_port 'z' pin;out_port 'rout' pin];
    'ipdm',
     [in_port 'x' pin;in_port 'y' pin;in_port 'in' pin;
      in_port 'reset' pin;in_port 'clk' pin;out_port 'yout' pin;
      out_port 'xout' pin;out_port 'z' pin;out_port 'rout' pin];
    'pdm',
     [cell_size 'n';in_port 'x' pin;in_port 'y' pin;in_port 'in' pin;
      in_port 'reset' pin;in_port 'clk' pin;out_port 'yout' pin;
      out_port 'xout' pin;out_port 'z' pin;out_port 'rout' pin]]

far
   |- !x y cin reset sum cout.
```

```
        far(x,y,cin,reset,sum,cout) =
        (?w1 w2 w3 w4 cab w5 w6 w7 w8.
          nand(x,y,w1) /\
          nand(x,cin,w2) /\
          nand(y,cin,w3) /\
          nand3(w1,w2,w3,w4) /\
          inv(w4,cab) /\
          nor(cab,reset,cout) /\
          nor3(x,y,cin,w5) /\
          inv(w5,w6) /\
          nand(cab,w6,w7) /\
          nand3(x,y,cin,w8) /\
          nand(w7,w8,sum))
  sa
    |- !x y sum clk reset.
       sa(x,y,sum,clk,reset) =
       (?cout cin.
         d_reg_nr(cout,clk,cin) /\ far(x,y,cin,reset,sum,cout))
  wff
    |- !in w out clk.
       wff(in,w,out,clk) =
       (?wb w1 w2 w3.
         inv(w,wb) /\
         nand(out,wb,w1) /\
         nand(in,w,w2) /\
         nand(w1,w2,w3) /\
         d_reg_nr(w3,clk,out))
  cell
    |- !x y in reset clk xout out rout.
       cell(x,y,in,reset,clk,xout,out,rout) =
       (?yi ab a b c outb.
         wff(y,reset,yi,clk) /\
         d_reg_nr(x,clk,xout) /\
         nand(x,yi,ab) /\
         inv(ab,a) /\
         sa(a,in,b,clk,reset) /\
         d_reg_wqb(reset,clk,rout,c) /\
         nand(b,c,outb) /\
         inv(outb,out))

  bps  |- !in out. bps(in,out) = (?w. inv(in,w) /\ inv(w,out))

  bcell
    |- !x y in reset clk yout xout z rout.
```

97

```
          bcell(x,y,in,reset,clk,yout,xout,z,rout) =
          bps(y,yout) /\ cell(x,y,in,reset,clk,xout,z,rout)

icell
  |- !xi yi zi ri clk yout xout z rout.
      icell(xi,yi,zi,ri,clk,yout,xout,z,rout) =
      (?yd xd zd rd.
        bcell(xi,yi,zi,ri,clk,yd,xd,zd,rd) /\
        d_reg_nr(yd,clk,yout) /\
        d_reg_nr(xd,clk,xout) /\
        d_reg_nr(zd,clk,z) /\
        d_reg_nr(rd,clk,rout))
ipdm
  |- (!x y in reset clk yout xout z rout.
      ipdm 0 x y in reset clk yout xout z rout =
      icell(x,y,in,reset,clk,yout,xout,z,rout)) /\
     (!n x y in reset clk yout xout z rout.
      ipdm(SUC n)x y in reset clk yout xout z rout =
      (?xi yi zi ri.
        ipdm n x y in reset clk yi xi zi ri /\
        icell(xi,yi,zi,ri,clk,yout,xout,z,rout)))
pdm
  |- (!n x y in reset clk yout xout z rout.
      pdm n x y in reset clk yout xout z rout =
      (?xi yi zi ri.
        ipdm n x y in reset clk yi xi zi ri /\
        bcell(xi,yi,zi,ri,clk,yout,xout,z,rout)))
```

D   Translated L program of Pipelined Multiplier

```
L:: TECH scmos
# hol2gdt: V.12.0 Built Dec 95, Anand V. Chavan

SCHEMATIC far ()
{
   IN    in1 ;
   IN    in2 ;
   IN    cin ;
   IN    reset ;
   OUT   sum ;
```

98

```
OUT    cout ;

INST nand nand[0];
WIRE nand[0].in[0] TO in1;
WIRE nand[0].in[1] TO in2;
SIG nand[0].out "w1";

INST nand nand[1];
WIRE nand[1].in[0] TO in1;
WIRE nand[1].in[1] TO cin;
SIG nand[1].out "w2";

INST nand nand[2];
WIRE nand[2].in[0] TO in2;
WIRE nand[2].in[1] TO cin;
SIG nand[2].out "w3";

INST nand3 nand3[0];
SIG nand3[0].in[0] "w1";
SIG nand3[0].in[1] "w2";
SIG nand3[0].in[2] "w3";
SIG nand3[0].out "w4";

INST inv inv[0];
SIG inv[0].in "w4";
SIG inv[0].out "cab";

INST nor nor[0];
SIG nor[0].in[0] "cab";
WIRE nor[0].in[1] TO reset;
WIRE nor[0].out TO cout;

INST nor3 nor3[0];
WIRE nor3[0].in[0] TO in1;
WIRE nor3[0].in[1] TO in2;
WIRE nor3[0].in[2] TO cin;
SIG nor3[0].out "w5";

INST inv inv[1];
SIG inv[1].in "w5";
SIG inv[1].out "w6";

INST nand nand[3];
SIG nand[3].in[0] "cab";
```

```
    SIG nand[3].in[1] "w6";
    SIG nand[3].out "w7";

    INST nand3 nand3[1];
    WIRE nand3[1].in[0] TO in1;
    WIRE nand3[1].in[1] TO in2;
    WIRE nand3[1].in[2] TO cin;
    SIG nand3[1].out "w8";

    INST nand nand[4];
    SIG nand[4].in[0] "w7";
    SIG nand[4].in[1] "w8";
    WIRE nand[4].out TO sum;
}

SCHEMATIC sa ()
{
    IN   x ;
    IN   y ;
    OUT  z ;
    IN   clk ;
    IN   reset ;

    INST d_reg_nr d_reg_nr[0];
    SIG d_reg_nr[0].in "co";
    WIRE d_reg_nr[0].clk TO clk;
    SIG d_reg_nr[0].out "ci";

    INST far far[0];
    WIRE far[0].in1 TO x;
    WIRE far[0].in2 TO y;
    SIG far[0].cin "ci";
    WIRE far[0].reset TO reset;
    WIRE far[0].sum TO z;
    SIG far[0].cout "co";
}

SCHEMATIC wff ()
{
    IN   in ;
    IN   w ;
    OUT  out ;
    IN   clk ;
```

```
    INST inv inv[0];
    WIRE inv[0].in TO w;
    SIG inv[0].out "wb";

    INST nand nand[0];
    WIRE nand[0].in[0] TO out;
    SIG nand[0].in[1] "wb";
    SIG nand[0].out "w1";

    INST nand nand[1];
    WIRE nand[1].in[0] TO in;
    WIRE nand[1].in[1] TO w;
    SIG nand[1].out "w2";

    INST nand nand[2];
    SIG nand[2].in[0] "w1";
    SIG nand[2].in[1] "w2";
    SIG nand[2].out "w3";

    INST d_reg_nr d_reg_nr[0];
    SIG d_reg_nr[0].in "w3";
    WIRE d_reg_nr[0].clk TO clk;
    WIRE d_reg_nr[0].out TO out;
}

SCHEMATIC cell ()
{
    IN   x ;
    IN   y ;
    IN   in ;
    IN   reset ;
    IN   clk ;
    OUT   xout ;
    OUT   out ;
    OUT   rout ;

    INST wff wff[0];
    WIRE wff[0].in TO y;
    WIRE wff[0].w TO reset;
    SIG wff[0].out "yi";
    WIRE wff[0].clk TO clk;

    INST d_reg_nr d_reg_nr[0];
    WIRE d_reg_nr[0].in TO x;
```

```
      WIRE d_reg_nr[0].clk TO clk;
      WIRE d_reg_nr[0].out TO xout;

      INST nand nand[0];
      WIRE nand[0].in[0] TO x;
      SIG nand[0].in[1] "yi";
      SIG nand[0].out "ab";

      INST inv inv[0];
      SIG inv[0].in "ab";
      SIG inv[0].out "a";

      INST sa sa[0];
      SIG sa[0].x "a";
      WIRE sa[0].y TO in;
      SIG sa[0].z "b";
      WIRE sa[0].clk TO clk;
      WIRE sa[0].reset TO reset;

      INST d_reg_wqb d_reg_wqb[0];
      WIRE d_reg_wqb[0].in TO reset;
      WIRE d_reg_wqb[0].clk TO clk;
      WIRE d_reg_wqb[0].out TO rout;
      SIG d_reg_wqb[0].out_b "c";

      INST nand nand[1];
      SIG nand[1].in[0] "b";
      SIG nand[1].in[1] "c";
      SIG nand[1].out "outb";

      INST inv inv[1];
      SIG inv[1].in "outb";
      WIRE inv[1].out TO out;
}

SCHEMATIC bps ()
{
   IN    in ;
   OUT   out ;

   INST inv inv[0];
   WIRE inv[0].in TO in;
   SIG inv[0].out "w";
```

```
    INST inv inv[1];
    SIG inv[1].in "w";
    WIRE inv[1].out TO out;
}

SCHEMATIC bcell ()
{
    IN    x ;
    IN    y ;
    IN    in ;
    IN    reset ;
    IN    clk ;
    OUT   yout ;
    OUT   xout ;
    OUT   z ;
    OUT   rout ;

    INST bps bps[0];
    WIRE bps[0].in TO y;
    WIRE bps[0].out TO yout;

    INST cell cell[0];
    WIRE cell[0].x TO x;
    WIRE cell[0].y TO y;
    WIRE cell[0].in TO in;
    WIRE cell[0].reset TO reset;
    WIRE cell[0].clk TO clk;
    WIRE cell[0].xout TO xout;
    WIRE cell[0].out TO z;
    WIRE cell[0].rout TO rout;
}

SCHEMATIC icell ()
{
    IN    xi ;
    IN    yi ;
    IN    zi ;
    IN    ri ;
    IN    clk ;
    OUT   yout ;
    OUT   xout ;
    OUT   z ;
    OUT   rout ;
```

103

```
    INST bcell bcell[0];
    WIRE bcell[0].x TO xi;
    WIRE bcell[0].y TO yi;
    WIRE bcell[0].in TO zi;
    WIRE bcell[0].reset TO ri;
    WIRE bcell[0].clk TO clk;
    SIG bcell[0].yout "yd";
    SIG bcell[0].xout "xd";
    SIG bcell[0].z "zd";
    SIG bcell[0].rout "rd";

    INST d_reg_nr d_reg_nr[0];
    SIG d_reg_nr[0].in "yd";
    WIRE d_reg_nr[0].clk TO clk;
    WIRE d_reg_nr[0].out TO yout;

    INST d_reg_nr d_reg_nr[1];
    SIG d_reg_nr[1].in "xd";
    WIRE d_reg_nr[1].clk TO clk;
    WIRE d_reg_nr[1].out TO xout;

    INST d_reg_nr d_reg_nr[2];
    SIG d_reg_nr[2].in "zd";
    WIRE d_reg_nr[2].clk TO clk;
    WIRE d_reg_nr[2].out TO z;

    INST d_reg_nr d_reg_nr[3];
    SIG d_reg_nr[3].in "rd";
    WIRE d_reg_nr[3].clk TO clk;
    WIRE d_reg_nr[3].out TO rout;
}
SCHEMATIC ipdm (INT hol_1_n = 4)
{
    IN    x;
    IN    y;
    IN    in;
    IN    reset;
    IN    clk;
    OUT   yout;
    OUT   xout;
    OUT   z;
    OUT   rout;
    INT   hol_1_i;
```

```
        hol_1_i = 0;
        WHILE(hol_1_i < hol_1_n) {
          INST icell icell[hol_1_i];

            IF(hol_1_i == 0) {
              WIRE icell[0].xi TO x;
              WIRE icell[0].yi TO y;
              WIRE icell[0].zi TO in;
              WIRE icell[0].ri TO reset;
            }

            WIRE icell[hol_1_i].clk TO clk;

            IF(hol_1_i != 0) {
              WIRE icell[hol_1_i-1].xout TO icell[hol_1_i].xi;
              WIRE icell[hol_1_i-1].yout TO icell[hol_1_i].yi;
              WIRE icell[hol_1_i-1].z TO icell[hol_1_i].zi;
              WIRE icell[hol_1_i-1].rout TO icell[hol_1_i].ri;
            }

            IF(hol_1_i == hol_1_n - 1) {
              WIRE icell[hol_1_i].yout TO yout;
              WIRE icell[hol_1_i].xout TO xout;
              WIRE icell[hol_1_i].z TO z;
              WIRE icell[hol_1_i].rout TO rout;
            }

            IF(hol_1_i != 0) {
            }
            hol_1_i++;
        }
    }

    SCHEMATIC pdm (INT hol_1_n = 1)
    {
        INT hol_1_i;
        STR hol_1_str1;
        STR hol_1_str2;
        IN    x ;
        IN    y ;
        IN    in ;
        IN    reset ;
        IN    clk ;
        OUT   yout ;
```

```
    OUT    xout ;
    OUT    z ;
    OUT    rout ;

    INST bcell bcell[0];
    SIG bcell[0].x "xi";
    SIG bcell[0].y "yi";
    SIG bcell[0].in "zi";
    SIG bcell[0].reset "ri";
    WIRE bcell[0].clk TO clk;
    WIRE bcell[0].yout TO yout;
    WIRE bcell[0].xout TO xout;
    WIRE bcell[0].z TO z;
    WIRE bcell[0].rout TO rout;

    INST ipdm ipdm[0];
    WIRE ipdm[0].x TO x;
    WIRE ipdm[0].y TO y;
    WIRE ipdm[0].in TO in;
    WIRE ipdm[0].reset TO reset;
    WIRE ipdm[0].clk TO clk;
    SIG ipdm[0].yout "yi";
    SIG ipdm[0].xout "xi";
    SIG ipdm[0].z "zi";
    SIG ipdm[0].rout "ri";
}
```

E   IRSIM Test Command File

```
h y reset
l clk
s 40
h clk
s 20
l clk
s 2

l reset y
h x in
s 40
h clk
```

```
s 20
l clk
s 2

l in
s 40
h clk
s 20
l clk
s 2

s 40
h clk
s 20
l clk
s 2

h y
s 40
h clk
s 20
l clk
s 2

x y
l x
s 40
h clk
s 20
l clk
s 2

h x
s 40
h clk
s 20
l clk
s 2

s 40
h clk
s 20
l clk
s 2
```

```
l x
h reset
s 40
h clk
s 20
l clk
s 2

x reset y x in
s 40
h clk
s 20
l clk
s 2

s 40
h clk
s 20
l clk
s 2

s 40
h clk
s 20
l clk
s 2

s 40
h clk
s 20
l clk
s 2

s 40
h clk
s 20
l clk
s 2

s 40
h clk
s 20
l clk
```

```
s 2

s 40
h clk
s 20
l clk
s 2

s 40
h clk
s 20
l clk
s 2

s 40
h clk
s 20
l clk
s 2

s 40
h clk
s 20
l clk
s 2

s 40
h clk
s 20
l clk
s 2

ana   clk reset y x in yout xout z rout
```

F   SPICE MOS Model Parameters

```
** SPICE file created for circuit padfinal
** Technology: scmos
**
.MODEL nfet NMOS (LEVEL=2.0 PHI=0.700000 TOX=3.9800E-08
+ XJ=0.200000U TPG=1
```

```
+ VTO=0.7794 DELTA=3.1470E+00 LD=1.8408E-07 KP=5.9259E-05
+ UO=683.0 UEXP=9.8530E-02 UCRIT=8.4200E+03 RSH=9.5900E+00
+ GAMMA=0.6033 NSUB=8.2550E+15 NFS=9.1000E+10 VMAX=5.1700E+04
+ LAMBDA=3.4430E-02 CGDO=2.405-10 CGSO=2.4051E-10
+ CGBO=3.4582E-10 CJ=1.24E-04 MJ=0.828 CJSW=5.68E-10
+ MJSW=0.324 PB=0.66)
* Weff = Wdrawn - Delta_W
* The suggested Delta_W is 2.0000E-09

.MODEL pfet PMOS (LEVEL=2 PHI=0.700000 TOX=3.9800E-08
+ XJ=0.200000U TPG=-1
+ VTO=-0.9373  DELTA=2.9690E+00 LD=1.5620E-07 KP=1.7153E-05
+ UO=197.7 UEXP=2.5700E-01 UCRIT=1.0910E+05 RSH=9.8190E-02
+ GAMMA=0.6667 NSUB=1.0080E+16 NFS=1.1000E+11 VMAX=9.9990E+05
+ LAMBDA=4.2200E-02 CGDO=2.0328E-10 CGSO=2.0328E-10
+ CGBO=4.1603E-10 CJ=3.38E-04 MJ=0.575 CJSW=2.48E-10
+ MJSW=0.289 PB=0.90)
* Weff = Wdrawn - Delta_W
* The suggested Delta_W is 2.0000E-09
```

G   SPICE Simulation Input file to Detect Clock Skew Between FF3 and FF4

```
** NODE: 0 = GND
** NODE: 1 = Vdd
** NODE: 2 = Error

VDD  1  0   DC   5V
Vclk 102  0   DC   PULSE (0V   5V   50ns   0  0  80ns  160ns)


R1    102   201   1380
M101  1     201   301   1   pfet    L=4.5U W=18.0U
M102  0     201   301   0   nfet    L=4.5U W=15.0U
M103  1     201   302   1   pfet    L=4.5U W=18.0U
M104  0     201   302   0   nfet    L=4.5U W=15.0U
M105  1     201   303   1   pfet    L=4.5U W=18.0U
M106  0     201   303   0   nfet    L=4.5U W=15.0U
M107  1     201   304   1   pfet    L=4.5U W=18.0U
M108  0     201   304   0   nfet    L=4.5U W=15.0U
M109  1     201   305   1   pfet    L=4.5U W=18.0U
```

```
M110   0      201    305    0    nfet      L=4.5U W=15.0U

R2     201    202    1380
M111   1      202    306    1    pfet      L=4.5U W=18.0U
M112   0      202    306    0    nfet      L=4.5U W=15.0U
M113   1      202    307    1    pfet      L=4.5U W=18.0U
M114   0      202    307    0    nfet      L=4.5U W=15.0U
M115   1      202    308    1    pfet      L=4.5U W=18.0U
M116   0      202    308    0    nfet      L=4.5U W=15.0U
M117   1      202    309    1    pfet      L=4.5U W=18.0U
M118   0      202    309    0    nfet      L=4.5U W=15.0U

R3     202    211    1380
M119   1      211    310    1    pfet      L=4.5U W=18.0U
M120   0      211    310    0    nfet      L=4.5U W=15.0U

R4     202    204    1380
M121   1      204    311    1    pfet      L=4.5U W=18.0U
M122   0      204    311    0    nfet      L=4.5U W=15.0U

R5     202    205    1380
M123   1      205    312    1    pfet      L=4.5U W=18.0U
M124   0      205    312    0    nfet      L=4.5U W=15.0U

R6     205    206    1380

R7     206    207    2760
M125   1      207    313    1    pfet      L=4.5U W=18.0U
M126   0      207    313    0    nfet      L=4.5U W=15.0U
M127   1      207    314    1    pfet      L=4.5U W=18.0U
M128   0      207    314    0    nfet      L=4.5U W=15.0U

R8     207    208    1380
M129   1      208    315    1    pfet      L=4.5U W=18.0U
M130   0      208    315    0    nfet      L=4.5U W=15.0U
M131   1      208    316    1    pfet      L=4.5U W=18.0U
M132   0      208    316    0    nfet      L=4.5U W=15.0U
M133   1      208    317    1    pfet      L=4.5U W=18.0U
M134   0      208    317    0    nfet      L=4.5U W=15.0U
M135   1      208    318    1    pfet      L=4.5U W=18.0U
M136   0      208    318    0    nfet      L=4.5U W=15.0U

R18    208    209    1380
M137   1      209    319    1    pfet      L=4.5U W=18.0U
```

```
M138    0      209    319    0    nfet     L=4.5U W=15.0U

R16     208    210    1380
M139    1      210    320    1    pfet     L=4.5U W=18.0U
M140    0      210    320    0    nfet     L=4.5U W=15.0U

R19     201    203    1380
M141    1      203    321    1    pfet     L=4.5U W=18.0U
M142    0      203    321    0    nfet     L=4.5U W=15.0U
M143    1      203    322    1    pfet     L=4.5U W=18.0U
M144    0      203    322    0    nfet     L=4.5U W=15.0U

R9      202    212    1380
M145    1      212    323    1    pfet     L=4.5U W=18.0U
M146    0      212    323    0    nfet     L=4.5U W=15.0U

R10     212    213    1380
M147    1      213    324    1    pfet     L=4.5U W=18.0U
M148    0      213    324    0    nfet     L=4.5U W=15.0U
M149    1      213    325    1    pfet     L=4.5U W=18.0U


R11     213    214    1380
M151    1      214    326    1    pfet     L=4.5U W=18.0U
M152    0      214    326    0    nfet     L=4.5U W=15.0U

R12     214    215    1380
M153    1      215    327    1    pfet     L=4.5U W=18.0U
M154    0      215    327    0    nfet     L=4.5U W=15.0U
M155    1      215    328    1    pfet     L=4.5U W=18.0U
M156    0      215    328    0    nfet     L=4.5U W=15.0U

R13     215    216    1380
M157    1      216    329    1    pfet     L=4.5U W=18.0U
M158    0      216    329    0    nfet     L=4.5U W=15.0U
M159    1      216    330    1    pfet     L=4.5U W=18.0U
M160    0      216    330    0    nfet     L=4.5U W=15.0U

R14     216    217    1380
M161    1      217    331    1    pfet     L=4.5U W=18.0U
M162    0      217    331    0    nfet     L=4.5U W=15.0U
M163    1      217    332    1    pfet     L=4.5U W=18.0U
M164    0      217    332    0    nfet     L=4.5U W=15.0U
M165    1      217    333    1    pfet     L=4.5U W=18.0U
```

```
M166  0     217   333   0   nfet    L=4.5U W=15.0U
M167  1     217   334   1   pfet    L=4.5U W=18.0U
M168  0     217   334   0   nfet    L=4.5U W=15.0U

R15   217   218   1380
M169  1     218   335   1   pfet    L=4.5U W=18.0U
M170  0     218   335   0   nfet    L=4.5U W=15.0U

R17   217   219   1380
M171  1     219   336   1   pfet    L=4.5U W=18.0U
M172  0     219   336   0   nfet    L=4.5U W=15.0U


.END
```

MISSION
OF
AFRL/INFORMATION DIRECTORATE (IF)

The advancement and application of Information Systems Science and Technology to meet Air Force unique requirements for Information Dominance and its transition to aerospace systems to meet Air Force needs.